ADT4HPC:
Algebraic Data Types for High-Performance Computing

Thaïs Baudon

August 2, 2024

This first page is temporary.

Phd Director Ludovic Henrio (CNRS, Lyon)

Co-advisors Laure Gonnord (Univ. Grenoble Alpes, Valence), Gabriel Radanne (Inria, Lyon)

Reviewers Gabriele Keller (Univ. Utrecht, The Netherlands), Pierre-Evariste Dagand (CNRS, Paris)

Other Jury Members Sandrine Blazy (Univ. Rennes); Sylvain Boulmé (Univ. Grenoble Alpes).

Invited Jury Member Lennart Augusston, (Göteborg, Sweden).

Contents

Chapter 1

Introduction

Initially present only in functional languages such as OCaml and Haskell, **Algebraic Data Types (ADTs)** have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions through **pattern matching**. Unfortunately, ADTs remain seldom used in low-level programming. One reason is that their increased convenience comes at the cost of abstracting away the exact **memory layout** of values. Yet high-performance [1] applications often rely on highly optimized representations of data in memory, which are hand-tuned by programmers to leverage very fine, low-level characteristics. For instance, red-black trees in the Linux kernel are a performance-intensive data structure for which it is crucial to minimize memory usage as much as possible. Their implementation therefore relies on a clever **bit-stealing** technique exploiting unused alignment bits in pointers. Such precise details are usually not exposed to programmers when it comes to ADT values. Even Rust, which tries to optimize data layout, severely limits control over memory representation.

The goal of this thesis is to let programmers specify highly optimized memory layouts for inductive data structures in a flexible and expressive way, while still enjoying high-level programming constructs such as ADTs and pattern matching to manipulate this data.

To this end, we propose a language dubbed **Ribbit** [2] which combines a *high-level* language, consisting of ADTs, pattern matching and basic manipulation of immutable values, with **memory types** specifying the precise memory layout of each high-level type, providing full control over the memory representation of values. We provide *formal semantics* of both (high-level and memory) languages which, together with **agreement criteria** stating the relationship between an ADT and a suitable memory layout, let us reason easily about values as they are represented in memory.

Compilation of high-level language constructs is heavily influenced by the memory representation of data. Traditional pattern matching compilation approaches, which emit a decision tree from a matrix of patterns, are geared towards a rather uniform data layout. Therefore, they are not suitable for compiling Ribbit programs, for which data layout is arbitrarily complex and variable. We propose a new pattern matching compilation approach based on **memory trees** which follows the specified memory type to emit an efficient decision tree suited to manipulating values with intricate memory layouts.

Aside from pattern matching, the compilation of **data constructors** – which is a non-issue for relatively simple memory layouts – becomes particularly challenging when data pieces are broken and scattered in memory. Even simple accessors might require constructing new values. This is the case for many **low-level representations** such as network packets, instruction sets, database data-structures, or aggressively packed representations. We propose a compilation algorithm for the Ribbit language which enables optimized compilation of any *morphism* between ADTs for arbitrary mangled memory representations, provides full synthesis of bijections between memory representations of the same type, and emits CFG-style programs with explicit memory allocation and full support for recursive types.

Our compilation algorithms are implemented in the Ribbit compiler prototype, and *proven correct* using the formal agreement criteria and semantics defined for Ribbit as a basis for establishing equivalence between high-level, memory-level and target programs.

[1] In this thesis, we assume a rather liberal interpretation of the term "High-Performance Computing".

[2] Named after a metaphor for two key aspects of manipulating data in memory: *knitting* and *frogging*. For crochet enthusiasts, frogging is the action of undoing the stitches: rip it, ripit, ribbit, ribbit...

The contents of this thesis are summarized below:

- Chapter 2 presents a collection of real-world memory layouts for ADTs, and shows how to use the Ribbit programming language to model them: by first declaring high-level types, then specifying their underlying memory representation.
- Chapter 3 presents the Ribbitulus, which formalizes Ribbit syntax and provides a simple type system with formal criteria to define valid memory representations of high-level types. We define a small-step semantics for both high-level and memory-level aspects of the language and exhibit a bisimulation between the two.
- Chapter 4 covers our pattern matching compilation approach for the Ribbit language, based on memory trees, which compiles high-level patterns to layout-aware decision trees according to a given memory type. We prove that our compilation algorithm is correct, i.e., that the emitted decision tree accurately identifies which pattern matches a value solely from its memory representation.
- Chapter 5 provides a complete compilation approach for the full Ribbit language, which emits code in a bespoke intermediate representation in destination-passing style. It handles all of the delicate situations which may arise during compilation of a high-level language with custom memory layouts, including implicit casts between different representations (so-called *memory isomorphisms* [3]), and recursive code emission. These algorithms are also proven correct, by showing that the target program's behavior is simulated by that of the source program.
- Chapter 6 describes some aspects of the Ribbit prototype compiler, which provides a practical validation of our approach [4]. In particular, the mutually recursive nature of the compilation algorithms manifesting memory isomorphisms makes their implementation especially delicate, and this chapter also details the necessary memoization techniques.
- Chapter 7 presents a preliminary experimental evaluation of our approach, with both static and runtime measurements of the decision trees emitted by our pattern matching compilation algorithm. It also demonstrates some of the performance impact of different memory representation choices.
- Chapter 8 explores related language-based approaches and other optimized memory representations.
- Chapter 9 concludes with some future work ideas.

This thesis includes and subsumes the following publications and research reports:

International Conference Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types, International Conference on Functional Programming, 2023 (Baudon, Radanne, and Gonnord 2023) – contains a limited version of the Ribbit language presented in Chapter 2, part of the calculus from Chapter 3, the pattern matching compilation approach described in Chapter 4, as well as its experimental evaluation covered in Chapter 7.

National Conference Knit&Frog: Pattern matching compilation for custom memory representations (doctoral session), french conference AFADL, (Baudon, Radanne, and Gonnord 2022a) – contains a preliminary version of Chapter 4.

Research reports Compiling Morphisms of Algebraic Data Types (Baudon, Radanne, and Gonnord 2024), to be presented at FProper2024 – contains a preliminary version of Chapter 5.

Tool Ribbit tool on Software Heritage (Baudon, Radanne, and Gonnord 2022b)

[3] Die-hard portmanteau enthusiasts may prefer the alternative term *memorphism*.
[4] As well as most of the dot graphs appearing in this thesis.

Part I

The Ribbit Language

Chapter 2

Memory Layout Zoo

As a first introduction to Algebraic Data Types and their memory representations, this chapter is a collection of *exhibits* showcasing a variety of real-world memory layouts for inductive data structures. It also serves as a high-level tour of our Ribbit programming language, illustrating its syntax and highlighting major features of its compiler.

2.1 My first Ribbit: Red-Black Trees

As our first exhibit, let us consider *Red-Black Trees*, a widely used inductive data structure. Its purpose is to demonstrate the use of Algebraic Data Types, some memory representation tricks, and how Ribbit allows to specify and compile them. We first describe their high-level type, then show various different memory layouts using the Ribbit language.

2.1.1 Algebraic Data Types for inductive data structures

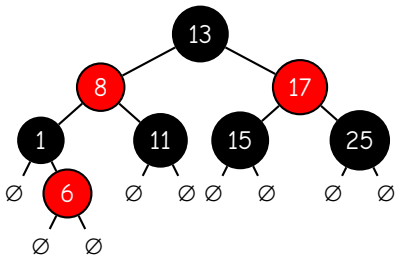
Red-Black Trees (Wikipedia 2024a) are a classic data structure from the family of search-trees whose main idea is to maintain an invariant on nodes based on two colors. Exhibit 1 shows a high-level implementation using Ribbit syntax. In Exhibit 1a, we first define the *sum type* `Color`, whose values are the two constant constructors `Red` and `Black`. We then define Red-Black Trees (RBTs) through the mutually recursive types `Node` for non-empty nodes, and `RBT` for trees themselves. A `Node` is a *product type* containing a color, a value of the *primitive type* of 64-bit-wide integers, and its left and right children. The type of trees `RBT` is a *sum type* with two cases: `Empty` and `Node`. This type enables the definition of `RBT` values. For instance, the red-black tree depicted in Exhibit 1b corresponds to the following term:

```
Node({c: Black, v: 13,
  l: Node({c: Red, v: 8,
    l: Node({c: Black, v: 1,
      l: Empty,
      r: Node({c: Red, v: 6, l: Empty, r: Empty})}),
    r: Node({c: Black, v: 11, l: Empty, r: Empty})}),
  r: Node({c: Red, v: 17,
    l: Node({c: Black, v: 15, l: Empty, r: Empty}),
    r: Node({c: Black, v: 25, l: Empty, r: Empty})})})
```

(a) Type definitions for Red-Black Trees.

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

// Nodes for Red-Black Trees
struct Node
{ c: Color, v: u64, // color and value
  l: RBT, r: RBT } // children

// Red-Black Trees
enum RBT { Empty, Node(Node) }
```

(b) Example of Red-Black Tree.



(c) Simple pattern matching: cardinal operation.

```
fn cardinal(x: RBT) -> u64 {
  match x {
    Empty => 0,
    Node({c:_, v:_, l, r}) => 1 + cardinal(l) + cardinal(r)
  }
}
```

(d) Less simple pattern matching: rebalancing operation.

```
fn balance(x: Node) -> RBT {
  match (x.c, x.v, x.l, x.r) {
    Black, z, Node({c: Red, v: y, l: Node({c: Red, v: x, l: a, r: b}), r: c}), d
    | Black, z, Node({c: Red, v: x, l: a, r: Node({c: Red, v: y, l: b, r: c})}), d
    | Black, x, a, Node({c: Red, v: z, l: Node({c: Red, v: y, l: b, r: c}), r: d})
    | Black, x, a, Node({c: Red, v: y, l: b, r: Node({c: Red, v: z, l: c, r: d})})
    => Node({c: Red, v: y, l: Node({c:Black,v:x,l:a,r:b}), r:Node({c:Black,v:z,l:c,r:d})}),
    _ => Node(x)
  }
}
```

(e) Toplevel Ribbit program manipulating RBTs.

```
let tree : RBT = Node({c: Black, v: 42, l: Empty, r: Empty});
let card : u64 = cardinal(tree);
let node : Node = {c: Red, v: card, l: tree, r: tree.r};
balance(node)
```

Exhibit 1: Algebraic Data Types and pattern matching for Red-Black Trees in ribbit.

Now that data types have been defined, we can write expressions and functions manipulating their inhabitants. A key language construct for manipulating ADT values is *pattern matching*. For instance, the `cardinal` function defined in Exhibit 1c takes a tree and returns its total number of nodes using pattern matching. In Ribbit, pattern matching is introduced by the keyword `match` and inspects a single value – in `cardinal`, it is `x` of type `RBT`. Pattern matching branches are enumerated in a list of the form `p => e` where `p` is a *pattern* and `e` an expression to evaluate when the value under scrutiny *matches* `p`. If its argument is of the same "shape" as the left-hand side of the rule, then the expression of the right-hand side (body) is evaluated. Moreover, patterns can be nested, and the right-hand-side expression can use named subterms. In our example, `Empty` yields a cardinal of `0` and `Node({c, v, l, r})` yields a cardinal of `1+cardinal(l)+cardinal(r)`.

Red-Black Trees famously rely on a fairly complex balancing step, which redistributes colors depending on the internal invariant of the data structure. Thanks to nested patterns and "or"-patterns, this step can be expressed very compactly using the pattern matching shown in Exhibit 1d. This pattern matching inspects the four record fields of a `Node` value `x`: its color `x.c`, value `x.v` and left and right subtrees `x.l` and `x.r`. The pattern of its second branch is a wildcard `_` which matches all values; it ensures that the pattern matching expression is *exhaustive*, i.e., that every possible value matches at least one pattern.

Such complex functions are expressible in a concise and safe way thanks to pattern matching; without this language construct, writing rebalancing code would be a clumsy and tedious task.

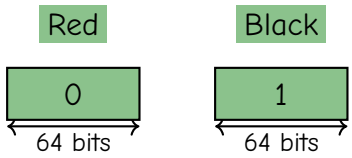
Finally, Ribbit provides basic features of a first-order functional language manipulating immutable data, namely `let`-bindings and function calls. Exhibit 1e shows a toplevel program which creates an `RBT` value, computes its cardinal and stores it in an intermediate `u64` value, builds a new `Node` value using the previous tree as its left child, its cardinal as its integer value and its right child as its own right child, and finally returns the balanced version of this RBT.

2.1.2 A naive memory representation for RBTs

Like most self-balancing trees, RBTs are a performance-intensive data structure. In a naive implementation, indirections in the memory representation limit locality, result in slow memory loads, cache misses, and slowdowns of several orders of magnitude. To achieve best possible performance, it is critical to pay attention to how values of our types are represented in the actual memory of the considered machine. We would nevertheless prefer to tweak the memory representation of data without mangling its *high-level* type, which provides nice data constructors and accessors that are close to intended type *semantics*. The Ribbit language provides detailed annotations called *memory types* to precisely describe the memory layout of each ADT. This memory layout specification language lets us capture a wide variety of popular representation techniques including bit-stealing, unboxing, aggressive struct packing, etc.

As our first foray into representation tweaking, we define a naive memory layout for RBTs. For each of our three ADTs `Color`, `Node` and `RBT`, we specify a memory type introduced by the keyword `represented as`. Let us first describe the memory type associated with `Color` in Exhibit 2.

(a) Graphical representation of memory contents



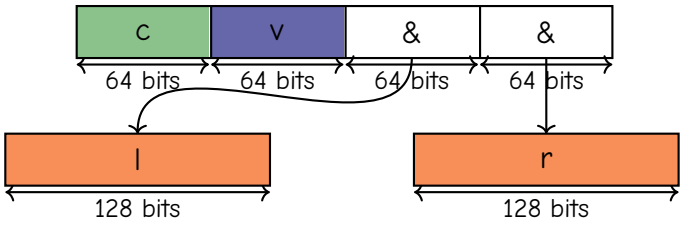
(b) Memory layout specification in ribbit

```
enum Color { Red, Black } represented as
split . { // empty path: inspect the whole value
  | 0 from Red => (0)<64>
  | 1 from Black => (1)<64>
}
```

Exhibit 2: A naive memory layout for `Color`.

The `Color` type is a sum type with two constructors `Red` and `Black`. To manifest the distinction between these constructors in memory, Ribbit provides the notion of `split`s. Split types of the form `split mpath {...}` indicate a choice between different memory layouts depending on the immediate stored at position `mpath` within the memory value. The memory position `mpath` is known as the split *discriminant position*, and consists of a sequence of operations such as pointer dereferences or memory accesses. In `Color`, the empty split discriminant position "`.`" inspects the entire memory value. The split then contains a list of branches, each containing an integer value dubbed its *discriminant value* and a pattern dubbed its *provenance* on its left-hand side, and a memory type on its right-hand side. Each branch indicates that high-level values which match its provenance (introduced by the keyword `from`) must be represented in memory using the layout on its right-hand side, which contains the specified discriminant value at the discriminant position. Here, we specify that `Red` is represented as the constant `0` encoded on 64 bits, denoted `(0)<64>`, and that `Black` is represented as the constant `1` encoded on 64 bits, denoted `(1)<64>`. This choice of layout is illustrated in Exhibit 2a using a graphical language consisting of sized boxes representing memory words. We will reuse this graphical language throughout this chapter.

(a) Graphical representation of memory contents corresponding to {c, v, l, r}



(b) Memory layout specification in Ribbit

```
struct Node {c:Color, v:u64, l:RBT, r:RBT} represented as
{{ (.c as Color), (.v as u64), &<64>((.l as RBT)), &<64>((.r as RBT)) }}
```

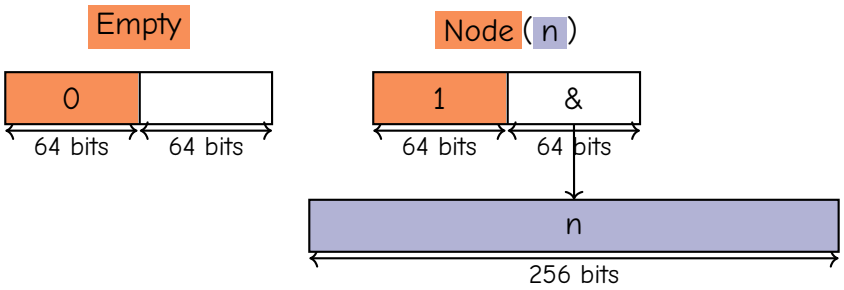
Exhibit 3: A naive memory layout for `Node`.

Let us now describe the memory layout used to represent `Node` values in Exhibit 3. The ADT `Node` is a product type which aggregates four fields together. We must represent each of these fields within the memory representation of their parent `Node` value. To do so, Ribbit provides the notion of *fragments*. Fragment types of the form `(path as MemTy)` indicate that the subterm at position `path` within the high-level value should be represented using the memory type `MemTy`. In `Node`, we represent the four record fields in a *struct* as follows:

- The color field corresponding to the subterm `.c` is encoded using the memory layout previously defined for `Color` and stored in the first field of the struct with the fragment `(.c as Color)`.
- The integer value corresponding to the subterm `.v` is encoded as a primitive 64-bit integer with the memory type `u64`. The resulting fragment `(.v as u64)` is placed in the second field of the struct. Each high-level primitive type has a memory counterpart which encodes its values using a standard encoding for the considered target system and architecture.
- For the left and right subtrees, corresponding to the subterms `.l` and `.r` respectively, we will use the memory layout defined for their type `RBT`. As we will see, this memory layout yields 128-bit wide memory values. In order to keep each struct field 64-bit wide, we will store both of these fragments behind a 64-bit wide pointer denoted `&<64>((.l as RBT))` for the left subtree and `&<64>((.r as RBT))` for the right subtree.

Note that Ribbit struct types do not include any implicit padding, unlike for instance C structs. This behavior is similar to "packed" struct types in LLVM IR, or to the `#[repr(packed)]` annotation in Rust. To ensure a given alignment for struct fields, the user can add explicit padding with uninitialized word types of a given size `l` denoted `_<l>`. As seen in Exhibit 3a, our graphical language represents pointers as sized words containing address bits denoted `&` and "pointing" to the memory contents stored at this address.

(a) Graphical representation of memory contents



(b) Memory layout specification in ribbit

```
enum RBT { Empty, Node(Node) } represented as
split .0 { // path .0: inspect the tag in the first field of the struct
  | 0 from Empty => {{ (0)<64>, _<64> }}
  | 1 from Node => {{ (1)<64>, &<64>((.Node as Node)) }}
}
```

Exhibit 4: A naive memory layout for `RBT`.

We can now define the memory layout for the `RBT` type shown in Exhibit 4. Every tree is represented in memory as a struct consisting of two 64-bit wide fields. The first field contains a *tag* which indicates whether the represented tree is `Empty` or a `Node` – we will therefore use its position `.0` as a split discriminant. The second field is left uninitialized for `Empty` trees; for non-empty trees `Node({c, v, l, r})`, it contains a 64-bit pointer to the memory representation of the root node's contents using the previously defined `Node` memory layout. Note that Ribbit allows recursive memory types: here, `Node` and `RBT` are two mutually recursive ADTs represented using two mutually recursive memory layouts.

Memory types allow us to specify the memory layout of each value down to a bit-precise level. In order to properly manipulate data represented using such custom memory layouts, the Ribbit compiler follows the structure of each memory type to emit appropriate target code. Two aspects of our input language require particular attention due to this variability in memory representation: pattern matching and data constructors. On our example, compiling the pattern matching in the `cardinal` function results in the *decision tree* depicted in Fig. 2.1. It consists of a single switch node which inspects the discriminant position `.0` of the `RBT` memory type to determine whether a given value `x` represents the `Empty` tree or a non-empty `Node`.

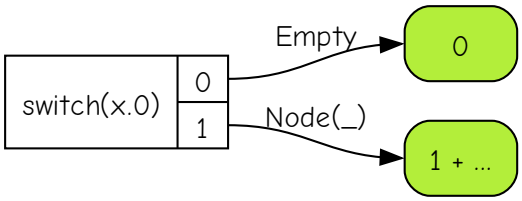


Figure 2.1: A decision tree for the pattern matching in the `cardinal` function.

As for the toplevel expression from Exhibit 1e, Ribbit compiles it to the low-level pseudo-code shown in Fig. 2.2. As before, this target code follows the specified memory layouts: for instance, Ribbit represents the high-level value `tree` of type `RBT` by building a memory value which closely follows the structure of its associated memory layout. Later on, to build the memory representation of the `node` value, whose right child is the previous `tree` value's right child, the `Tree` memory type is used to determine that the subterm `tree.r` is located at position `.1.*.3` (i.e., the fourth field of the struct pointed to by the second field of the root struct) within the memory representation of `tree`.

```
let tree = {{ (1)<64>, &<64>({{ // Node
  (1)<64>,  // c: Black
  (42)<64>, // v: 42
  &<64>({{ (0)<64>, _<64> }}), // l: Empty
  &<64>({{ (0)<64>, _<64> }})  // r: Empty
}})}};
let card = cardinal(tree);
let node = {{ (1)<64>, &<64>({{ // Node
  (0)<64>,  // c: Red
  (card)<64>, // v: card
  tree, // l: tree
  tree.1.*.3  // r: tree.r
}})}};
return balance(node);
```

Figure 2.2: Target code emitted by Ribbit for the source program of Exhibit 1e following the memory layouts defined in Exhibits 2 to 4.

Naturally, compiling high-level programs to layout-aware target code is only possible when each memory type adequately represents its associated ADT: we say that a high-level type must *agree* with its memory layout. Here, the memory type specified for `RBT` in Exhibit 4b is *valid* and *agrees* with the `RBT` ADT because:

- All subterms of the high-level type `RBT` (here, `.Node`) are properly accounted for in the memory type with a fragment (here, `(.Node as Node)`).
- All constructors of the high-level type (here, `Empty` and `Node`) are properly accounted for in the memory type with a split branch.
- Split branches are all distinguishable from each other: their left-hand sides contain different values (here, 0 and 1), corresponding to distinct constructors (here, `Empty` and `Node`).

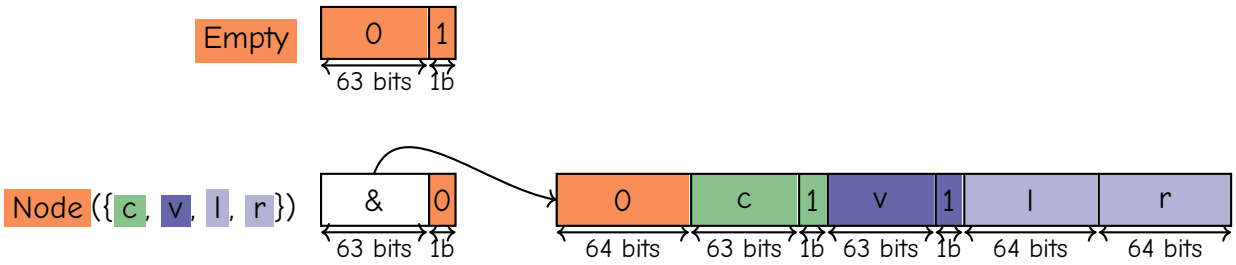
The formalization of these *agreement criteria* is also a key contribution of this thesis, which we will detail in Section 3.2.

2.1.3 OCaml-like representation of RBTs

While correct, the naive memory layout defined in the previous section is not particularly efficient: every tree takes up 128 bits of memory space, even in the `Nil` case; in addition, every node introduces a layer of indirection, with disastrous consequences on performance. Such outrageous memory layouts are unlikely to be found in real-world languages, even in garbage-collected high-level languages where performance is not necessarily the main focus.

In this section, we take a first look at such a language, and show that Ribbit is expressive enough to model its internal memory representation. The memory types defined in Exhibit 5 represent RBTs in memory similarly to the OCaml runtime (Minsky and Madhavapeddy 2021).

(a) Graphical representation of memory values



(b) Ribbit specification with memory types

```
enum Color { Red, Black } represented as
/* Composite word with two specified bit ranges */
_<64> with [0:1] : (1)<1>
      with [1:63] : split . {
        | 0 from Red => (0)<63>
        | 1 from Black => (1)<63>
      }

struct Node {c:Color, v:u63, l:RBT, r:RBT} represented as
&<64>({{
  (0)<64>,
  (.c as Color),
  (.v as _<64> with [0:1] : (1)<1> with [1:63] : u63),
  (.l as RBT),
  (.r as RBT)
}}) with [0:1] : (0)<1>

enum RBT { Empty, Node(Node) } represented as
split .[0:1] {
  | 1 from Empty =>
    _<64> with [0:1] : (1)<1> with [1:63] : (0)<63>
  | 0 from Node(_) =>
    &<64>({{ (0)<64>,
      (.Node.c as Color),
      (.Node.v as _<64> with [0:1]:(1)<1> with [1:63]:u63),
      (.Node.l as RBT), (.Node.r as RBT)
    }}) with [0:1] : (0)<1>
}
```

Exhibit 5: The OCaml layout for RBTs. Ribbit is also able to automatically generate these memory types from its *generic* OCaml representation scheme.

In OCaml, all types are represented uniformly, for instance as 64-bit words on 64-bit architectures. This uniformity allows for an easier implementation of polymorphism, and keeps the garbage collector happy. The lowest bit of every memory value is used as a *tag* to distinguish between unboxed values (i.e., not stored in a pointer or other container) and pointers; standard immediates are therefore restricted to 63 bits rather than 64. Since pointers are word-aligned, their lowest bit is always zero; conversely, we tag every immediate value by setting its lowest bit to one. To avoid needing to box the integer value of each RBT node, we have slightly altered the high-level type so that nodes carry 63-bit integers of type `u63`.

This tagging scheme requires us to separately specify the contents of distinct bit ranges within the same memory word. To express such bit-precise memory layouts, Ribbit provides the notion of *composite words*, denoted `MemTy with [o:l]:MemTy' with ...`. Such composite words consist of a base memory type `MemTy` onto which we add an arbitrary number of *bit range specifications*. Each bit range specification `with [o:l]:MemTy'` indicates that the range of `l` bits starting at offset `o` within `MemTy` follows the memory layout `MemTy'`. Of course, this is only valid if these bits are not already used by the

base layout `MemTy`. For instance, the memory representation of the `Empty` constructor of `RBT` is a 64-bit word whose lowest bit is set to `1` and whose remaining 63 higher bits encode the constant `0`, denoted `_<64> with [0:1]:(1)<1> with [1:63]:(0)<63>`.

In OCaml, unit constructors are represented as tagged immediates corresponding to their unique identifier among unit constructors of the same type. For instance, `Red` and `Black` are represented as `_<64> with [0:1]:(1)<1> with [1:63]:(0)<63>` and `_<64> with [0:1]:(1)<1> with [1:63]:(1)<63>` respectively. Constructors with arguments, on the other hand, are represented as pointers to a struct whose first field contains their unique identifier among non-unit constructors and whose other fields encode their arguments. For instance, non-empty trees of the form `Node{c, v, l, r}` are represented as a pointer to a struct whose first field encodes `0` on 64 bits and whose next four fields encode `c`, `v`, `l` and `r`. This adds up to a total of six 64-bit memory words, including the pointer.

Ribbit also provides *generic memory representations* – such as the OCaml representation – which automatically generate a memory type from a given ADT according to a generic scheme. For our example, we could have specified the `Color`, `Node` and `RBT` memory layouts of Exhibit 5 with `represented by caml` rather than by writing specific memory types by hand. We will detail the available generic representations in Section 2.6.

2.1.4 Linux-like custom memory layout for RBTs

The OCaml layout removes a layer of indirection compared to the naive representation, but is hampered by its uniform nature. For instance, a full word is used to store the color, even though it technically requires only one bit. Let us look at a highly optimized representation of RBTs originally found in the Linux kernel to model device trees (Torvalds 2023). The representation is hand-tuned to take as little space as possible by embedding the color in pointer alignment bits. This optimization is known as *bit-stealing*. The original type definition in C from the x86-64 Linux kernel source version 6.9.2-gentoo is shown in Exhibit 6. The general red-black tree type (`struct rb_root`) does not include the values (in our case, these would be 64-bit integers) carried by each node: instead, users must define their own structs containing an `rb_root` and the element type of their choice. The type of RBT nodes `struct rb_node` is a struct containing three 64-bit fields. Its two last fields are pointers to the left and right subtrees. Its first field `__rb_parent_color` is more unusual: it is a 64-bit word containing both a pointer to the parent node (or `NULL`) and the color of the current node stored in its lowest bit. As we can see in Exhibit 6, many accessors which would be trivial to implement on a uniform memory representation such as that of OCaml require hand-written code due to the extremely irregular memory layout. Note in particular the parent pointer access, which requires some careful bit masking before dereferencing.

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
/* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root {
    struct rb_node *rb_node;
};
```

Exhibit 6: Type definition – excerpt from /include/linux/rbtree_types.h (Torvalds 2023).

```
#define rb_parent(r)   ((struct rb_node *)((r)->__rb_parent_color & ~3))
```

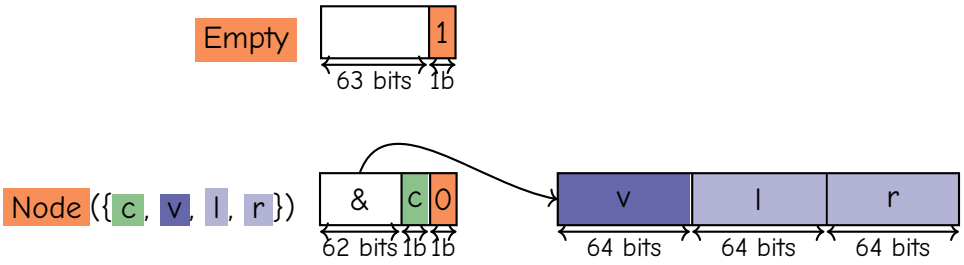
Exhibit 6: RBT constructors – excerpt from /include/linux/rbtree.h (Torvalds 2023).

```
#define RB_RED          0
#define RB_BLACK        1

#define __rb_parent(pc)    ((struct rb_node *)(pc & ~3))

#define __rb_color(pc)     ((pc) & 1)
#define __rb_is_black(pc)  __rb_color(pc)
#define __rb_is_red(pc)    (!__rb_color(pc))
#define rb_color(rb)       __rb_color((rb)->__rb_parent_color)
#define rb_is_red(rb)      __rb_is_red((rb)->__rb_parent_color)
#define rb_is_black(rb)    __rb_is_black((rb)->__rb_parent_color)
```

Exhibit 6: RBT constructors – excerpt from /include/linux/rbtree_augmented.h (Torvalds 2023).

Our goal is now to capture this intricate memory layout in the Ribbit language, without having to alter the high-level types `Color`, `Node` and `RBT`. Due to the limited scope of the high-level portion of Ribbit – in particular, we do not handle mutable values and restrict ourselves to ADTs without back-references to parent values – we eschew the parent pointer of the original version. Instead, we model a possible version of such an intricate memory layout for immutable data. We combine the clever bit-stealing of the original memory layout with OCaml-style pointer tagging, using a **`split`** with composite words to model the multi-purpose 64 bits of `__rb_parent_color`. We also hard-code a 64-bit integer element type which we store in a struct alongside left and right child trees. The resulting memory layout is shown in Exhibit 7.

(a) Graphical representation of memory values



(b) Ribbit specification with memory types

```
enum Color { Red, Black } represented as
split . {
  | 0 from Red => (0)<1>
  | 1 from Black => (1)<1>
}

struct Node { c: Color, v: u64, l: RBT, r: RBT } represented as
&<64>({{ (.Node.v as u64), (.Node.l as RBT), (.Node.r as RBT) }})
with [1:1] : (.Node.c as Color)

enum RBT { Empty, Node(Node) } represented as
split .[0:1] {
  | 1 from Empty => _<64> with [0:1] : (1)<1>
  | 0 from Node =>
    &<64>({{ (.Node.v as u64), (.Node.l as RBT), (.Node.r as RBT) }})
    with [0:1] : (0)<1> with [1:1] : (.Node.c as Color)
}
```

Exhibit 7: The Linux memory layout for red-black trees.

Even though we have specified a new, complex memory layout for red-black trees, the original

ADTs `Color`, `Node` and `RBT` are unchanged. As such, the rest of the program shown in Exhibit 1 is still valid and will work unmodified. A similar change of representation in another context would often require a significant code rewrite. In particular, writing low-level data manipulation code for an optimized memory representation is rather painful and error-prone, requiring delicate handling of memory contents with fine-grained operations for every data access. For instance, the original C code manipulating red-black trees in the Linux kernel was shown in Exhibit 6 for a selection of simple accessors. In Exhibit 8, we show *one simple case* of the rebalancing operation defined in Exhibit 1d. All case analysis is implemented by hand using masks and manual dereferencing for every manipulation of the corresponding data. On top of being tedious and error-prone, this also obscures program semantics compared to pattern matching.

```
gparent = rb_red_parent(parent);

tmp = gparent->rb_right;
if (parent != tmp) {        /* parent == gparent->rb_left */
        if (tmp && rb_is_red(tmp)) {
                /*
                 * Case 1 - node's uncle is red (color flips).
                 *
                 *       G            g
                 *      / \          / \
                 *     p   u  -->   P   U
                 *    /            /
                 *   n            n
                 *
                 * However, since g's parent might be red, and
                 * 4) does not allow this, we need to recurse
                 * at g.
                 */
                rb_set_parent_color(tmp, gparent, RB_BLACK);
                rb_set_parent_color(parent, gparent, RB_BLACK);
                node = gparent;
                parent = rb_parent(node);
                rb_set_parent_color(node, parent, RB_RED);
                continue;
        }
```

Exhibit 8: One case of the rebalancing operation – excerpt from /lib/rbtree.c (Torvalds 2023).

Ribbit spares the user the tedious task of writing such low-level code by hand thanks to its compiler, which automatically emits target code taking into account the specific memory layout of each piece of data. For instance, given the memory layouts specified in Exhibit 7 and the `balance` function defined in Exhibit 1d, the Ribbit pattern matching compiler emits the decision tree shown in Fig. 2.3.

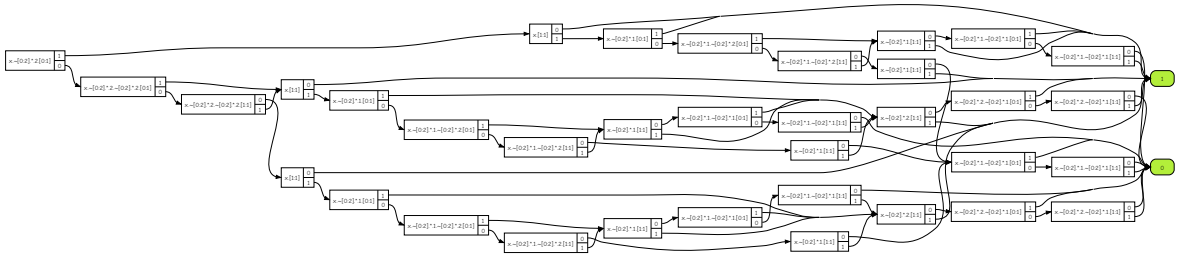


Figure 2.3: Decision tree emitted by the Ribbit compiler for the pattern matching in the `balance` function with the Linux-like RBT memory layout.

Decision trees are a common target for pattern matching compilation. Since this pattern matching is

at the core of a performance-sensitive data structure, we naturally want it to be as efficient as possible. Many pattern matching implementations come with clever techniques to output optimized decision trees (Kosarev, Lozov, and Boulytchev 2020; Maranget 2008; Sestoft 1996). Unfortunately, these are designed for terms that directly reflect the structure of their algebraic data types. In the context of Ribbit, a decision tree consists of *switch nodes* which inspect locations in memory determined by the specified memory layout, and of leaves (shown in light green) which return the identifier of the pattern corresponding to the considered memory contents. For instance, the root node of the decision tree in Fig. 2.3 is a switch on the memory location `x.~[0:2].*.2.[0:1]`, which indicates whether the right subtree of `x` is `Empty` or a `Node`. It first masks off the lowest bits (containing the tag and color), dereferences the underlying pointer, accesses the second struct field to get the representation of `x.l`, and finally extracts its lowest bit which corresponds to its tag. The design of new algorithms to compile pattern matching in the presence of memory types is a second contribution of this thesis, which will be described in Chapter 4.

2.2 A fine layout for a simple ADT: Zarith-like integers

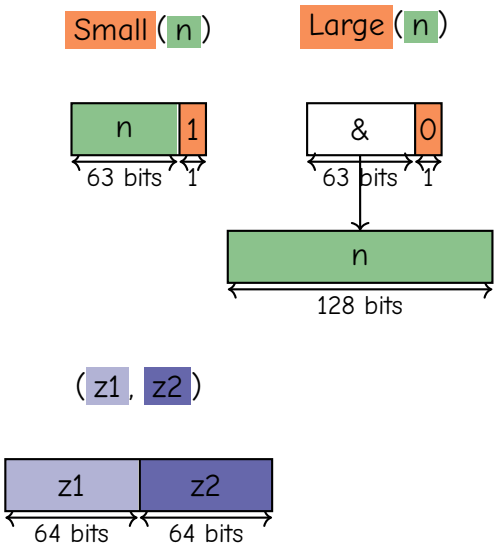
While complex inductive structures such as red-black trees are an important use case of ADTs and pattern matching, some simpler data types also benefit from these tools and from custom memory layouts. We now dive deeper into Ribbit's compilation process using such an example: Zarith (Leroy and Miné 2010), an OCaml library for arbitrary-precision integers. To speed up computations, integers in Zarith are either "small", represented as unboxed OCaml integers and using usual instructions, or "large", stored on the heap and manipulated using the GMP Bignum library. The choice of memory layout made in Zarith is not expressible in OCaml. Instead, it is implemented using unsafe operations via the C foreign function interface (FFI). Exhibit 9 shows an excerpt of the Zarith C implementation, including the conversion function `ml_z_of_nativeint`.

```c
/*
A z object x can be:
- either an ocaml int
- or a block with abstract or custom tag and containing:
  . a 1 value header containing the sign Z_SIGN(x) and the size Z_SIZE(x)
  . Z_SIZE(x) mp_limb_t

Invariant:
- if the number fits in an int, it is stored in an int, not a block
- if the number is stored in a block, then Z_SIZE(x) >= 1 and
the most significant limb Z_LIMB(x)[Z_SIZE(x)] is not 0
*/


/* a sign is always denoted as 0 (+) or Z_SIGN_MASK (-) */
#ifdef ARCH_SIXTYFOUR
#define Z_SIGN_MASK 0x8000000000000000
#define Z_SIZE_MASK 0x7fffffffffffffff
#else
#define Z_SIGN_MASK 0x80000000
#define Z_SIZE_MASK 0x7fffffff
#endif

#if Z_CUSTOM_BLOCK
#define Z_HEAD(x)   (*((value*)Data_custom_val((x))))
#define Z_LIMB(x)   ((mp_limb_t*)Data_custom_val((x)) + 1)
#else
#define Z_HEAD(x)   (Field((x),0))
#define Z_LIMB(x)   ((mp_limb_t*)&(Field((x),1)))
#endif
#define Z_SIGN(x)   (Z_HEAD((x)) & Z_SIGN_MASK)
#define Z_SIZE(x)   (Z_HEAD((x)) & Z_SIZE_MASK)

/* ... */

CAMLprim value ml_z_of_nativeint(value v)
{
  intnat x;
  value r;
  Z_MARK_OP;
  x = Nativeint_val(v);
#if Z_USE_NATINT
  if (Z_FITS_INT(x)) return Val_long(x);
#endif
  Z_MARK_SLOW;
  r = ml_z_alloc(1);
  if (x > 0) { Z_HEAD(r) = 1; Z_LIMB(r)[0] = x; }
  else if (x < 0) { Z_HEAD(r) = 1 | Z_SIGN_MASK; Z_LIMB(r)[0] = -x; }
  else Z_HEAD(r) = 0;
  Z_CHECK(r);
  return r;
}
```

Exhibit 9: Zarith's C side – excerpt from caml_z.c (Leroy and Miné 2010).

We can readily describe the memory layout of Zarith integers in Ribbit. In Exhibit 10, we define the type `Zint` of Zarith-like integers and its memory layout following the previous specification. As Ribbit does not interface with external libraries (yet), we model the GMP Bignum integer type with 128-bit primitive integers `i128`.

(a) Graphical representation of memory contents



(b) Implementation in Ribbit

```
enum Zint { Small(i63), Large(i128), }
represented as
split .[0:1] { // inspect the lowest bit
  | 1 from Small(_) =>
    _<64> with [1:63]:(.Small as i63)
  | 0 from Large(_) =>
    &<64>(.Large as i128)
}

struct Zpair(Zint, Zint); represented as
{{ (.0 as Zint), (.1 as Zint)
```

Exhibit 10: Memory layouts for Zarith-like integers and their pairs.

A `Zint` value is represented in the `Small` case as a 64-bit word with its lowest bit set to `1`, and the higher 63-bits encoding the actual integer. The `Large` case is represented as a 64-bit pointer to a 128-bit word encoding its value, with the lowest bit set to `0` to distinguish it from the `Small` case. As before, we use fragments to specify the memory representation of the integer subterm in both branches. Note that we did not explicitly specify the discriminant value in the split's branches. Ribbit is indeed able to infer it from the split's discriminant position `.[0:1]` and from the constant (0 or 1) associated with each branch. It will automatically add the bit range specification `with [0:1]:(1)<1>` to the `Small(_)` branch type and `with [0:1]:(0)<1>` to the `Large(_)` branch type. We also define the ADT `Zpair`, which is a product type grouping two `Zint` values together. Its memory layout simply represents it as a struct containing both of its fields' representations.

Let us now focus on how Ribbit compiles data manipulation code according to this Zarith-like memory layout. In Fig. 2.4, we define a function `leq` comparing the two fields of a `Zpair` value. Using pattern matching, we determine the head constructors of these two `Zint` values. For this example, let us assume that primitive operations `i63.<=` and `i128.<=` are available to compare raw 63-bit and 128-bit integers respectively. If both head constructors are identical, we simply compare their identically-sized integer values using the appropriate comparison operator. Otherwise, we must extend the `Small` field's integer value to 128 bits in order to compare it with the `Large` field's value. We denote this primitive cast operation with `(i128)(n)`.

```
1  fn leq(p : Zpair) -> Bool {
2    match p {
3      (Small(n1), Small(n2)) => n1 i63.<= n2,
4      (Large(n1), Large(n2)) => n1 i128.<= n2,
5      (Small(n1), Large(n2)) => (i128)(n1) i128.<= n2,
6      (Large(n1), Small(n2)) => n1 i128.<= (i128)(n2)
7    }
8  }
```

Figure 2.4: The `leq` function on Zarith-like integer pairs.

To compile the `leq` function, we must emit target code which performs the following tasks:

1. inspect memory contents to determine the head constructor of both `Zint` fields;
2. extract the raw integer values from both fields;

3. perform the operation corresponding to the right-hand side of the matched pattern.

The Ribbit compiler automatically emits low-level code carrying out these tasks, using the specified memory type to determine the precise location of each piece of data. Its output is shown in Fig. 2.5. Its general structure is that of a decision tree inspecting each field's discriminant value, corresponding to the pattern matching on `p`. Its leaves contain instructions corresponding to the expression on each pattern's right-hand side. In all four cases, we must extract both fields' raw integer values from their respective memory locations before comparing them.

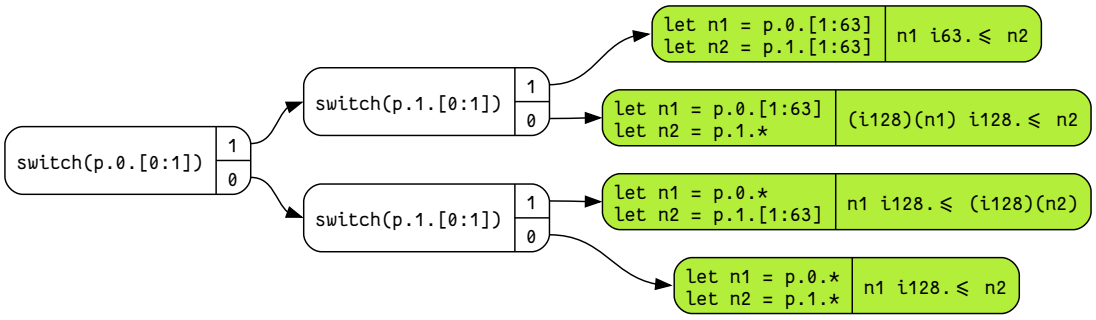


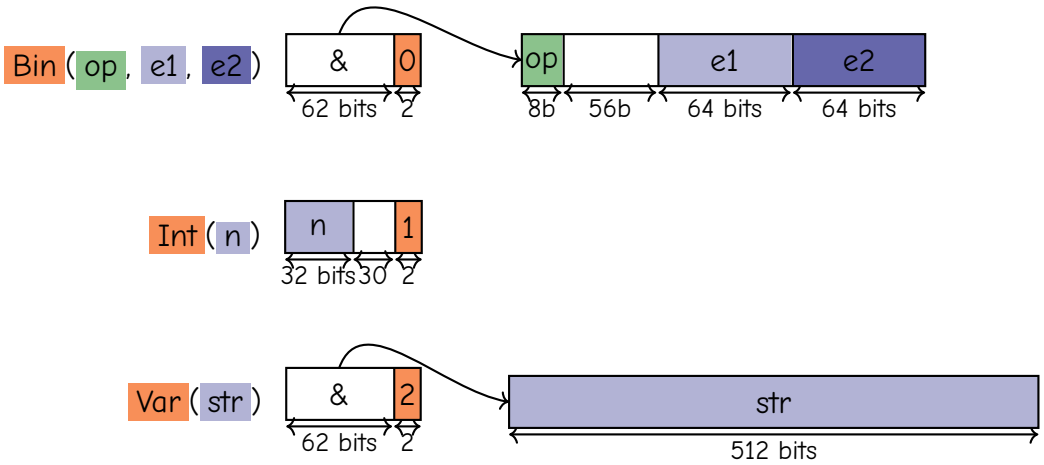
Figure 2.5: Output of the Ribbit compiler for the `leq` function.

2.3 Irregular memory layouts: arithmetic expressions

All memory layouts we have seen so far were *regular*, in that the hierarchy of splits and fragments closely followed that of the represented ADT. Even in complex layouts such as the Linux-like layout for red-black trees, each sum type constructor was represented by exactly one split branch and each product type field by exactly one fragment. In this section, we introduce *irregular* layouts which rearrange these components in new ways. As we will see, this irregularity impacts data manipulation code and compilation.

Consider the type `ExpAST` of simple arithmetic expressions on 32-bit integers defined in Exhibit 11. Such an expression is either a variable name `Var(str)` with the string `str` modeled as a 512-bit integer, a 32-bit integer constant `Int(n)`, or a binary operation `Bin(op, e1, e2)` where `op` is either `Plus` or `Mult`. We specify a naive memory layout reminiscent of abstract syntax trees for the `ExpAST` ADT. First off, `Op` values are represented on 8 bits similar to a C enum. Every expression is represented on 64 bits, using the two lowest bits as a split discriminant to distinguish between the three possible head constructors `Var`, `Int` and `Bin`. Assuming a 64-bit architecture with word-aligned machine pointers, the two lowest bits of pointers are indeed unused, allowing us to use them to store the split discriminant in the `Var` and `Bin` cases. We store the 32-bit integer value of an `Int` expression into the 32 highest bits of its 64-bit word. `Var` expressions are simply represented as a pointer to the pseudo-string (`i512`) argument. Finally, we represent a `Bin` expression as a pointer to a struct containing the representations of its operation identifier and of its two operands. In order to maintain 64-bit alignment, we explicitly pad the first field with 32 unspecified bits `_<32>`.

(a) Graphical representation of memory contents

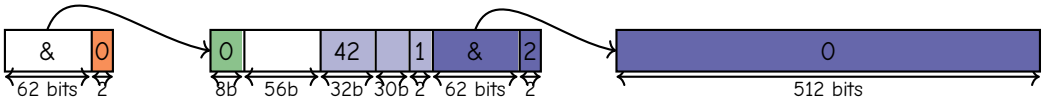


(b) Specification in Ribbit

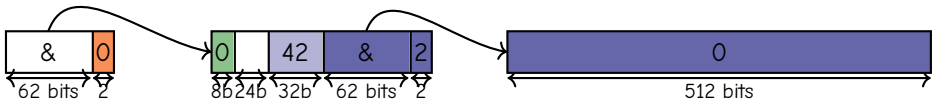
```
type String = i512; represented as i512
enum Op { Plus, Mult } represented as
split . {
  | 0 from Plus => (0)<8>
  | 1 from Mult => (1)<8>
}
enum ExpAST { Var(String), Int(i32), Bin(Op, ExpAST, ExpAST) } represented as
split .[0:2] {
  | 0 from Bin(_) =>
    &<64>({{ (.Bin.0 as Op), _<56>, (.Bin.1 as ExpAST), (.Bin.2 as ExpAST) }})
  | 1 from Int(_) => _<64> with [32:32]:(.I as i32)
  | 2 from Var(_) => &<64>((.V as String))
}
```

Exhibit 11: Arithmetic expressions and their AST-like memory representation.

Even though the `ExpAST` layout is correct, it is quite wasteful. For instance, consider the expression `Bin ( Plus , Int(42) , Var(0) )`. We currently represent it as a pointer to a struct whose fields take up a total of 192 bits, pictured below.



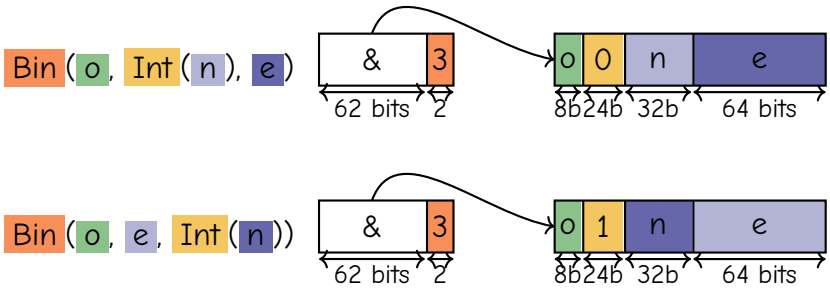
To save space, we could inline the 32-bit integer value into the unused space next to the `Op` value. By reducing the remaining padding to 24 bits, we would get a 128-bit-wide struct as shown below.



More generally, `Bin` expressions with at least one `Int` operand can be compressed to save memory by unboxing their integer values. We define this optimized memory layout `ExpOpt` in Exhibit 12. The ADT modeling arithmetic expressions and its auxiliary types `String` and `Op` are unchanged from the previous type `ExpAST`. The memory representations of standalone `Int` terms, of `Var` terms and of `Plus`

expressions with no integer operands are also unchanged. Vertical bars | in split provenances denote "or"-patterns – for instance, `Bin(_)|Var(_)` matches `Bin` and `Var` values.

(a) Graphical representation of memory contents for the new optimized `Plus` case



(b) Specification in Ribbit

```
type String = i512; represented as i512
enum Op { Plus, Mult } represented as
split . { 0 from Plus => (0)<8> | 1 from Mult => (1)<8> }

enum ExpOpt { Var(String), Int(i32), Plus(ExpOpt, ExpOpt) } represented as
split .[0:2] {
  | 0 from Bin(_, Bin(_)|Var(_), Bin(_)|Var(_)) =>
    &<64>({{ (.Bin.0 as Op), _<56>, (.Bin.1 as ExpOpt), (.Bin.2 as ExpOpt) }})
  | 1 from Int(_) =>
    _<64> with [32:32] : (.Int as i32)
  | 2 from Var(_) =>
    &<64>((.Var as String))
  | 3 from Bin(_, Int(_), _) | Bin(_, _, Int(_)) =>
    &<64>(split .1 {
      | 0 from Bin(_, Int(_), _) =>
        {{ (.Bin.0 as Op), (0)<24>, (.Bin.0.Int as i32), (.Bin.1 as ExpOpt) }}
      | 1 from Bin(_, Bin(_)|Var(_), Int(_)) =>
        {{ (.Bin.0 as Op), (1)<24>, (.Bin.1.Int as i32), (.Bin.0 as ExpOpt) }}
    })
}
```

Exhibit 12: Optimized memory layout for arithmetic expressions.

To model binary expressions with inlined integer operands, we add a new branch to the toplevel split with the previously unassigned discriminant value 3. These expressions are represented as a pointer (whose two lowest bits are set to 3) to a struct containing their `Op` field on 8 bits as before, followed by a 24-bit tag determining whether the integer operand appears in first or second position, the inlined 32-bit value of said integer operand, and finally the remaining operand encoded on 64 bits.

While the `ExpOpt` memory layout saves space compared to `ExpAST`, it pervasively impacts the compilation process. Indeed, `Int` values are now represented differently depending on the context in which they appear – either as a standalone expression or as an operand of a `Bin` expression. Conversely, the memory representation of a `Bin` value may follow two different split branches depending on its operands. As a consequence, seemingly simple patterns and values now require complex code to properly manipulate data, which is why such optimizations are usually only done by programmers when absolutely necessary (such as extremely performance-sensitive code). Ribbit alleviates this by automatically emitting layout-aware target code, making such complex layouts completely transparent to client code.

As an example of a program whose compilation is complicated by irregular memory layouts, consider the `eval` function shown in Fig. 2.6. It reduces all arithmetic expressions which can be evaluated without knowing variable operands' values. When both operands of a binary operation are integers, it computes the result of the operation and returns an `Int` expression representing this integer value (lines 6 and 7).

```
1  fn eval(e : ExpOpt) -> ExpOpt {
2    match e {
3      Int(_) | Var(_) => e,
4      Bin(op, e1, e2) => match (eval(e1), eval(e2)) {
5        Int(n1), Int(n2) => match op {
6          | Plus => Int(n1 + n2),
7          | Mult => Int(n1 * n2)
8        },
9        e1', e2' => Bin(op, e1', e2')
10     }
11   }}
```

Figure 2.6: User implementation of `eval`.

We now give a high-level view of Ribbit's global compilation procedure, which we formalize in Chapter 5, and of some intermediate program representations used in its implementation, which we present in more detail in Chapter 6.

As a first compilation step, we desugar the body of `eval` to its normalized form shown in Fig. 2.7. We use an explicitly typed A-Normal Form representation in which patterns contain no variables and subterms are instead referred to by their positions. For instance, `e.Bin.1` accesses the first operand of the binary expression `e`. Such accesses are only valid under the right pre-conditions.

The final compiled output is shown in Fig. 2.8 as a Control Flow Graph in *Destination Passing Style* (Shaikhha et al. 2017): rather than returning its result, it fills a destination memory location `d` with appropriate contents. Corresponding parts of `eval` in normalized source and final target code are highlighted in matching colors. As we have seen in previous examples, we compile pattern matching to a decision tree consisting of switch nodes inspecting relevant parts of memory to determine the shape of the considered input data. In our general compilation procedure for Ribbit, we integrate each such decision tree into the output control-flow graph structure. For our `eval` example, the three source matches on lines 2, 6 and 11 each correspond to one switch node in the compiled CFG inspecting the adequate discriminant location.

Beyond pattern matching, the compilation of other elements of the source program, namely data constructors and accessors, is complicated by the irregular memory layout. Consider the code which extracts both operands of the `Bin` head constructor (on line 4). Given our `ExpOpt` memory layout, this is not a straightforward memory access: the location of the subterms `e.Bin.1` and `e.Bin.2` depends on the precise shape of their parent value `e`. Furthermore, either of these subexpressions may be *unboxed* (i.e., encoded as their integer value when their head constructor is `Int`), meaning their data is not necessarily stored as `ExpOpt` within `e`. We must therefore *rebuild* the `ExpOpt` representation of both operands from their pieces extracted from `e` to be used as arguments to `eval` calls. To do so, the generated code allocates 64 bits for both `e1` and `e2`, inspects the discriminant `e.*.1` which determines how their various pieces are laid out within `e`, and finally fills their contents accordingly. More formally, we have synthesized an *isomorphism* between the existing representation of `e.Bin.1` (e.g., an inlined integer value within `e`) and its desired representation (a standard `ExpOpt`).

As we have seen, simple accessors at the source level might require us to emit code that allocates memory and performs various other operations. Consider now line 20, where we return the value `Bin(op, e1',e2')`. Following the `ExpOpt` memory layout, it is split into three cases: depending on whether the head constructor of `e1'` or `e2'` is `Int`, we must build this value differently. Note that this decision is factorized with the previous pattern matching. Again, in all cases, the emitted code allocates and places every bit in memory. To do so, we once more had to synthesize a morphism from the available memory representations of `e1'` and `e2'` as standalone expressions to their representation as operands of a `Bin` operation. This novel compilation procedure that can destruct and rebuild values and manifest isomorphisms between representations is a large contribution of this thesis described in Chapter 5.

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

```
fn eval(e : ExpOpt) -> ExpOpt {
  match e {
    Bin(_, _, _) =>
      let e1:Exp = e.Bin.1; let e2:Exp = e.Bin.2;
      let e1':Exp = eval(e1); let e2':Exp = eval(e2);
      match (e1', e2') {
        Int(_), Int(_) =>
          let n1:i32 = e1'.Int;
          let n2:i32 = e2'.Int;
          let op:Op = e.Bin.0;
          match op {
            Plus =>
              let n:i32 = n1 + n2;
              Int(n),
            Mult =>
              let n:i32 = n1 * n2;
              Int(n)
          },
        _, _ =>
          Bin(op, e1', e2')
      },
    _ => e,
}}
```

Figure 2.7: Normalized representation of `eval`.

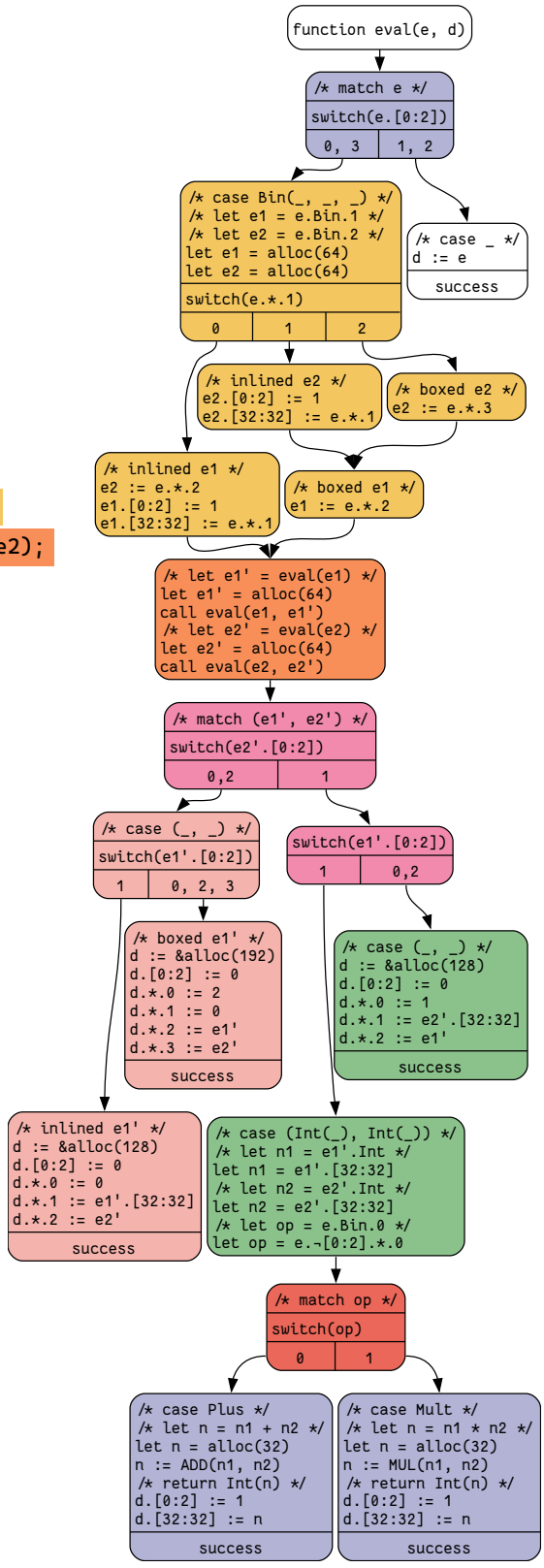


Figure 2.8: Simplified CFG after compiling `eval`. For pedagogic and readability purposes, code has been simplified (block sinking, variable renaming, simple constant propagation).

2.4 Recursive data constructors: simple and packed linked lists

So far, we only considered one memory layout at once for each ADT. However, some situations call for different memory layouts for the same data at different times. This usually requires converting data from one layout to the other, which can be quite complex. In particular, some combinations of recursive memory layouts require the compiler to emit recursive target code. In this section, we show how Ribbit handles such situations using a list type with two different memory layouts. These types will also be used in Chapter 3 as a running example to illustrate our formalization of the Ribbit language.

Consider the type of lists of 32-bit integers, which we define as the ADT `List` in Exhibit 13. We first define a simple memory layout representing it as simply-linked lists with one level of indirection per element. Notice how the second field of the struct used to represent the `Cons` case is not 64-bit aligned: indeed, Ribbit will pack struct fields together without inserting any padding nor reordering fields. One way to restore alignment would be to insert explicit 32-bit padding between the two fields. Even so, the resulting memory layout would not be particularly efficient: each element of a list results in one new level of indirection and uses 128 bits of memory (32 bits for the `u32` value itself, 32 padding bits and 64 bits to represent the next link).

A more efficient way to represent lists of 32-bit integers on a 64-bit architecture would be to pack two elements per level of indirection. In Exhibit 13, we demonstrate such a layout with the `PairList` ADT, whose inhabitants are exactly those of `List`. Its memory type is a split with three branches, distinguishing between empty, single-element and multiple-element lists. Notice how in the `Cons(_, Cons(_))` case, we represent two list elements in the same struct, allowing us to maintain 64-bit alignment without wasting any space.

```
enum List { Nil, Cons(u32, List) } represented as
split .[0:1] {
  | 1 from Nil => _<64> with [0:1]:(1)<1>
  | 0 from Cons(_) =>
    &<64>({{ (.Cons.0 as u32), (.Cons.1 as List) }})
    with [0:1]:(0)<1>
}

enum PairList { Nil, Cons(u32, PairList) } represented as
split .[0:2] {
  | 0 from Nil => _<64> with [0:2]:(0)<2>
  | 1 from Cons(_, Nil) =>
    _<64> with [0:2]:(1)<2> with [2:32]:(.Cons.0 as u32)
  | 2 from Cons(_, Cons(_)) =>
    &<64>({{
      (.Cons.0 as u32), (.Cons.1.Cons.0 as u32),
      (.Cons.1.Cons.1 as PairList)
    }}) with [0:2]:(2)<2>
}

fn single_to_double(l : List) -> PairList { l }

fn double_to_single(l : PairList) -> List { l }
```

Exhibit 13: A recursive ADT for lists and two possible memory layouts.

On their own, neither of these two memory layouts is particularly noteworthy: Ribbit handles value construction and pattern matching compilation for either `List` or `PairList` similar to previous examples. However, the fact that the `List` and `PairList` ADTs describe the same high-level data structure allows us to convert values from one layout to the other, as is done by the `single_to_double` and `double_to_single` functions.

Such user-level conversion code leverages the same feature of Ribbit that we used in Section 2.3

to compile the `eval` function with the `ExpOpt` irregular memory layout, that is, its ability to exhibit isomorphisms between different memory representations of the same data. In this specific case, we are dealing with two fundamentally different arrangements of the inductive structure of lists: in order to convert a `List` to a `PairList` or vice-versa, we must walk the entire recursive structure to fuse blocks into pairs. As we will see in Chapter 5, the Ribbit compiler handles such situations by emitting recursive target code manifesting the isomorphism between the two representations. Figure 2.9 shows the emitted recursive code for the `single_to_double` conversion function.

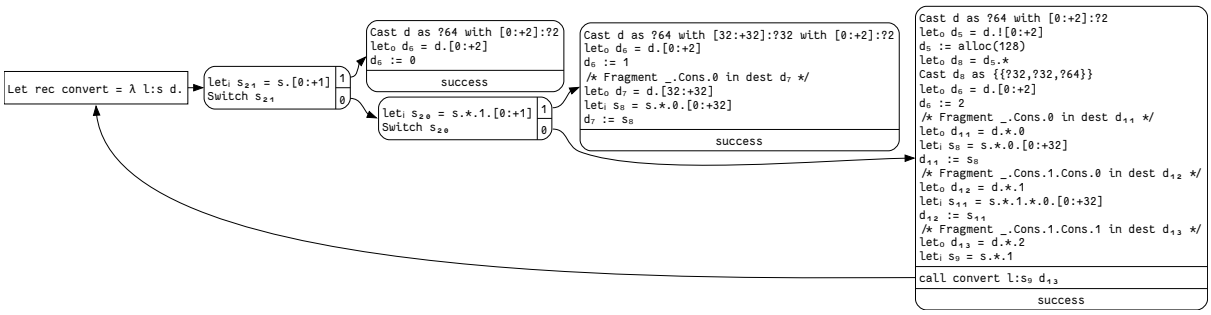


Figure 2.9: Generated code for rebuilding linked lists. `s` is the input list and `d` the destination location.

2.5 Mangled primitives: RISC-V instruction set

In this section, we consider a restricted version of the 32-bit RISC-V assembly language. We will use Ribbit to specify as a memory layout the encoding described in the instruction set (ISA) documentation (Waterman et al. 2019). A distinctive feature of this memory layout is the way it encodes integer values: the immediate operand of some instructions is broken down into its individual bits, which are then scattered across the memory representation of the whole instruction. Here, we show how such mangled primitive values can be expressed and manipulated with Ribbit.

Before expressing it in Ribbit, let us describe the subset of the RISC-V instruction set we wish to capture. It consists of four instructions: `add`, `addi`, `sw`, and `jal`, whose semantics are given in Fig. 2.10. A RISC-V machine has 32 registers, `x0` to `x31` (encoded on 5 bits). As shown in Fig. 2.11, RISC-V 32-bit instructions have different formats based on their addressing mode. Further characteristics of our four instructions are in Fig. 2.10.

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

Figure 2.10: Instruction semantics and encoding, excerpt.

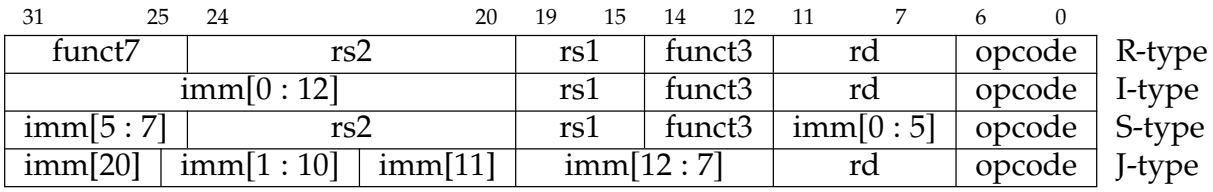


Figure 2.11: RISC-V Core instruction format, excerpt. "rs1,2" are source registers, "rd" a destination register. "imm[n]" denotes the n-th bit of imm. "imm[$o : \ell$]" means "ℓ bits starting from o in the binary representation of imm".

Already, we see complications: in general, instruction characteristics (type, instruction name, involved registers, etc.) are spread over `opcode`, `funct3` and `funct7`, which are stored non-consecutively. Moreover, the latter two are sometimes not present in the 32-bit instruction value. Immediates are particularly mangled, and cannot be readily extracted from the binary representation. For our particular (simple) subset :

1. the four instructions are distinguishable from their `opcode` *only*, stored in bits 0 to 7 inclusive;
2. the destination registers of `add` and `addi` are at the same location, bits 7 to 11;
3. the immediate value (`imm`) for the `sw` instruction is split and stored in two bit ranges: bits 7 to 11 and 25 to 31;
4. the 20-bit immediate value for the `jal` instruction can be recovered from bits 12 to 31 but we need to *rebuild* this immediate from four separate bit ranges.

We now use Ribbit to model RISC-V registers and instructions with the `Reg` and `Instr` ADTs and memory layouts in Exhibit 14. The `Reg` ADT is a simple sum type enumerating the 32 available RISC-V registers, which we wish to represent similarly to a C enum: each register `Xi` should be represented as the constant `i` (for instance, `X2` is the constant integer "3"). For this purpose, Ribbit provides a predefined representation which we request with the keywords `represented by C`. It will automatically find the minimal required width to encode each possible value – in our case, 5 bits since there are 32 registers and $2^5 = 32$ – and assign each constructor to its identifier encoded on this width. For the `Reg` type, this corresponds to the following memory type: `split . { 0 from X0 => (0)<5> | ... | 31 from X31 => (31)<5> }`. Section 2.6 will provide more detail on memory representations predefined by Ribbit.

```
/* C-like enum stored on 5 bits */
enum Reg { X0, X1, ..., X31 } represented by C

enum Instr {
  Add(Reg, Reg, Reg), // add rd, rs1, rs2
  Addi(Reg, Reg, i12), // addi rd, rs1, imm12
  Jal(Reg, i20), // jal rd, imm20
  Sw(Reg, Reg, i12), // sw rs1, rs2, imm12
} represented as
split .[0:7] { // inspect the opcode stored in the 7 lowest bits
  | 0x13 from Addi(_, _, _) =>
    _<32> with [0:7] : (0x13)<7> // opcode constant
    with [7:5] : (.Addi.0 as Reg) // register operand rd
    with [12:3] : (0)<3> // funct3 constant
    with [15:5] : (.Addi.1 as Reg) // register operand rs1
    with [20:12] : (.Addi.2 as i12) // immediate operand imm
  | 0x23 from Sw(_, _, _) =>
    _<32> with [0:7] : (0x23)<7> // opcode constant
    with [7:5] : (.Sw.2.[0:5] as i5)  // 5 lowest bits of immediate operand
    with [12:3] : (2)<3> // funct3 constant
    with [15:5] : (.Sw.0 as Reg) // register operand rs1
    with [20:5] : (.Sw.1 as Reg) // register operand rs2
    with [25:7] : (.Sw.2.[5:7] as i7)   // 7 highest bits of immediate operand
  | 0x6f from Jal(_, _) =>
    _<32> with [0:7] : (0x6f)<7> // opcode constant
    with [7:5] : (.Jal.0 as Reg) // register operand rd
    /* scattered pieces of the immediate operand imm */
    with [12:7] : (.Jal.1.[11:7] as i7) with [20:1] : (.Jal.1.[10:1] as i1)
    with [21:10] : (.Jal.1.[0:10] as i10) with [31:1] : (.Jal.1.[19:1] as i1)
  | 0x33 from Add(_, _, _) =>
    _<32> with [0:7] : (0x33)<7> // opcode constant
    with [7:5] : (.Add.0 as Reg) // register operand rd
    with [12:3] : (0)<3> // funct3 constant
    /* register operands rs1 and rs2 */
    with [15:5] : (.Add.1 as Reg) with [20:5] : (.Add.2 as Reg)
    with [25:7] : (0)<7> // funct7 constant
}
```

Exhibit 14: Ribbit ADTs and memory types for 32-bit RISC-V registers and instructions.

Our `Instr` ADT for RISC-V 32-bit instructions is a sum type with four constructors corresponding to the four instructions in our subset. Their operands are either registers or integers of various widths. Following the RISC-V specification shown in Fig. 2.11, we encode them on 32 bits and distinguish them using their opcode stored in the 7 lowest bits. The `Instr` memory type is therefore a split whose discriminant is the memory location `.[0:7]` and whose four branches each describe a 32-bit composite word.

The memory layout of some instructions is relatively simple. For instance, in the `Addi` branch, we partition the 32-bit uninitialized word `_<32>` into five distinct bit ranges. The bit ranges `[0:7]` and `[12:3]` correspond to the `opcode` and `funct3` constants: their contents are set to the adequate constant words `(0x13)<7>` and `(0)<3>` following Fig. 2.10. The three remaining bit ranges each store an operand, as specified in the S-type line of Fig. 2.11. For both register operands `.Addi.0` and `.Addi.1` and the 12-bit immediate operand `.Addi.2`, we use a fragment to specify the adequate memory type `Reg` or `i12`.

Other instructions such as `Jal` have a more intricate representation. As in the previous branch, we partition the 32-bit word into several bit ranges, with `[0:7]` containing the opcode constant and `[7:5]` storing the register operand `.Jal.0`. However, the 20-bit immediate operand `.Jal.1` is broken

down into four parts which are stored non-consecutively in four separate bit ranges. We ask Ribbit to represent portions of this `i20` integer value separately using fragments with the following syntax: `(.Jal.1.[o:l] as il)`, where `[o:l]` denotes the `l` consecutive bits starting from offset `o` in the "standard" `i20` representation of this immediate.

Now that types and layouts have been defined, we can focus on high-level code manipulating RISC-V instructions and, most importantly, its compilation to correct target data manipulation code using Ribbit. Given a 12-bit integer value `imm`, consider the data constructor `Sw(X1, X2, imm)`. From a high-level perspective, this is a simple constructor: using a typical representation for such a value, we would simply allocate an adequate amount of memory, then encode `X1`, `X2` and `imm` as integers at their assigned positions. Our representation, however, is not so straightforward: since `imm` is stored non-consecutively, we need to break it down into two pieces read from two different positions in its own memory representation. In essence, we need to synthesize code manifesting the *isomorphism* between the previous representation of `imm` (here, a standard `i12`) and the representation embedded in `Instr` (two pieces at stored at positions `.[7:5]` and `.[25:7]` within the instruction). As we have seen in previous sections of this chapter, such implicit recombination of subterms is common in the context of embedded and low-level memory representations. A simple struct flattening and reordering already exhibits a similar behavior.

Given this data constructor, the Ribbit compiler emits the low-level code shown in Fig. 2.12, which builds the memory representation of `Sw(X1, X2, imm)` by:

1. allocating enough memory – here, 32 bits – to hold this value in a new memory location `x`;
2. initializing the parts of `x` corresponding to constant parts of the desired value – here, its `opcode` and `funct3` constants and its two register operands;
3. reading both parts (bit ranges `[0:5]` and `[7:5]`) of the (non-constant) immediate operand from its memory location `imm`, and writing their contents to their adequate positions in `x`.

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

Figure 2.12: Code building the memory representation of `Sw(X1, X2, imm)` in the memory location `x`.

Let us now consider a more complex example of a source program manipulating `Reg` and `Instr` values. The RISC-V instruction set manual contains a standard extension for compressed instructions (Waterman et al. 2019, chapter RISCV-C) which defines a compressed 16-bit encoding for some instructions. Whether a given 32-bit instruction can be compressed according to this standard depends on the rules specified for this operation. Usually, it requires immediate operands to be small enough to fit in reduced space, and register operands to belong to the eight "most popular" registers `X8` to `X15` inclusive.

In Exhibit 15, we define the `is_compressible` function which determines whether a given `Instr` value corresponds to a compressible 32-bit RISC-V instruction. Its two companion functions `is_nonzero_register` and `is_popular_register` are predicates on `Reg` values whose names are self-explanatory. The definitions of these three functions rely on boolean operations (equality, comparison, etc.) which are built into Ribbit.

```
fn is_nonzero_register(r : Reg) -> Bool {
  match r {
    X0 => False,
    _ => True
  }
}

fn is_popular_register(r : Reg) -> Bool {
  match r {
    X8 | X9 | X10 | X11 | X12 | X13 | X14 | X15 => True,
    _ => False
  }
}

fn is_compressible(x : Instr) -> Bool {
  match x {
    Jal(X1, off) => off < 4096,
    Add(rd, rs1, rs2) => rd == rs1 && is_nonzero_register(rs1) && is_nonzero_register(rs2),
    Addi(rd, rs, imm) => rd == rs && is_nonzero_register(rs) && imm < 64,
    Sw(rbase, roff, imm) =>
      is_popular_register(rbase) && is_popular_register(roff) &&
      imm.[0:5] == 0 && imm.[10:2] == 0,
    _ => False,
  }
}
```

Exhibit 15: Function determining whether a given 32-bit instruction can be compressed into a 16-bit one, in Ribbit syntax.

Ribbit's compilation algorithm, detailed in Chapter 5, can emit the control flow graph depicted in Fig. 2.13, which:

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

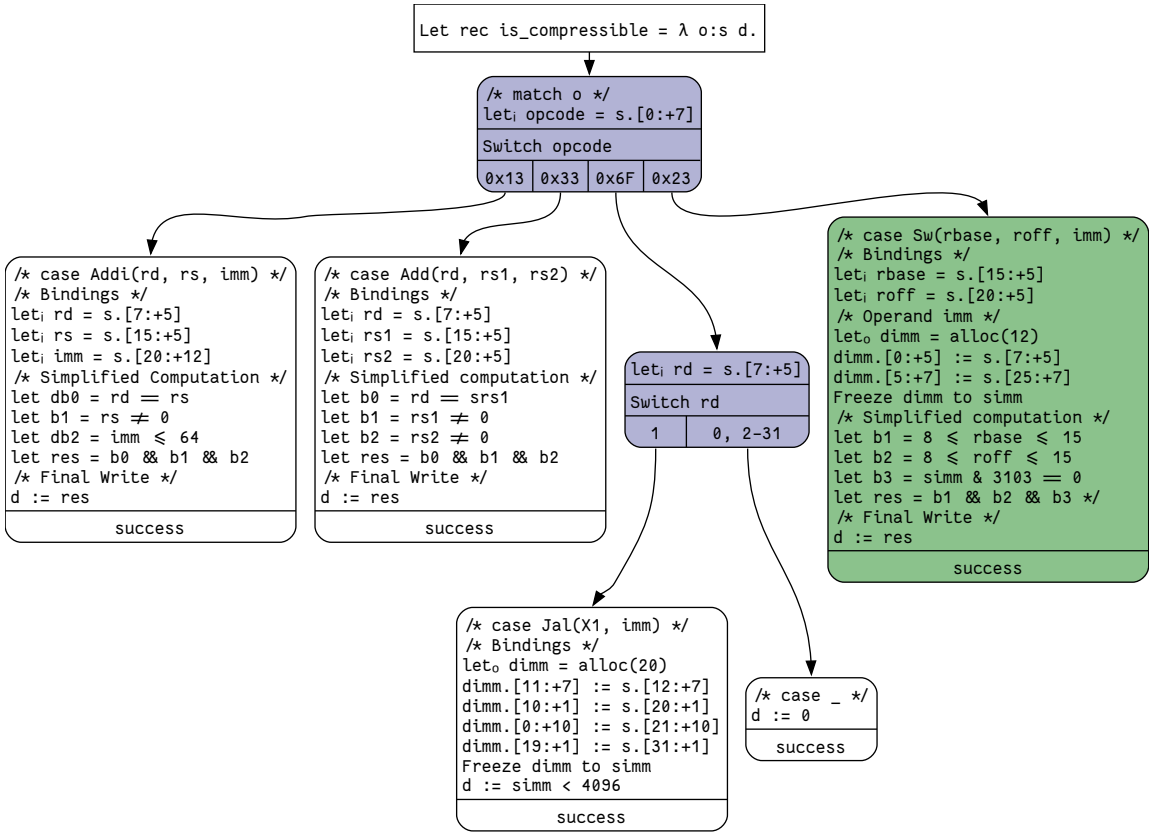


Figure 2.13: Simplified CFG for **`is_compressible`**. From the input `i`, it identifies the head constructor using the 7 lowest bits, then extracts subterms such as destination and source registers for `Add` or the 12-bit `imm` for `Sw` (in bold), and finally stores the result in `dest`.

2.6 Generic representations of ADTs in mainstream languages

In previous sections, we described a variety of memory layouts for specific ADTs. However, it is not always possible nor desirable to specify the memory layout of each individual data type. Most programming languages have their own standard way to represent data in memory, with varying degrees of flexibility and customization by the user. Similarly, Ribbit provides a (very limited) selection of predefined representations which follow common rules to represent any given ADT. In this section, we explore the generic memory representations of several mainstream languages and show how they can be modeled in Ribbit.

Note that most information presented in this section is potentially incomplete and obtained from a variety of sources ranging from official language references to folklore knowledge of specific compiler mechanics. Other sources include unofficial language documentation, various blog posts, as well as manual inspection of intermediate representations emitted by compilers.

2.6.1 OCaml

OCaml is a garbage-collected, functional programming language. Such languages were the first to natively support ADTs and pattern matching. Accordingly, OCaml features a rich type system which includes ADTs in the form of tuples and records for product types and of sum types dubbed "variants". For instance, the red-black tree ADTs from Section 2.1 can be modeled as the following OCaml types:

```
type color = Red | Black

type rbt =
  | Empty
  | Node of (color * int * rbt * rbt)
```

Exhibit 16: Red-black trees in OCaml.

As OCaml is a quite high-level language, it does not offer users precise control over low-level aspects such as data layout. Furthermore, garbage collection requires memory contents to follow a somewhat predictable pattern. The OCaml runtime therefore represents values according to a uniform memory layout which we describe below, using Minsky and Madhavapeddy (2021) as our main source.

Every OCaml value is either *unboxed*, i.e., represented on a single machine word whose lowest bit is set to 1 to tag it as an unboxed value, or *boxed*, i.e., represented as a pointer (whose lowest bit is always 0 due to address alignment) to a struct containing a header followed by data fields. Exhibit 17 depicts these two kinds of values in memory for a 64-bit platform.

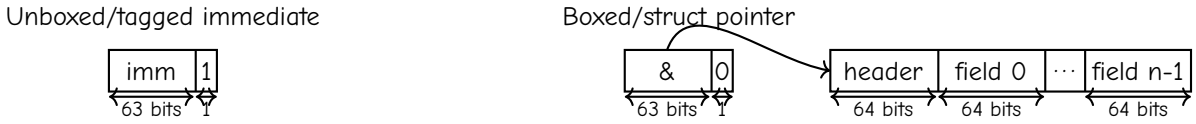


Exhibit 17: OCaml generic representation for 64-bit architectures.

More precisely, unboxed values are used to represent small enough primitive values (e.g., 63-bit integers of type `int`) and unit constructors of sum types (e.g., `Red`, `Black` and `Empty` constructors from Exhibit 16).

All other values, such as floating-point numbers, tuples and non-unit value constructors, are represented using the boxed layout. For instance, consider a value of the type `rbt` defined in Exhibit 16 of the form `Node(c, v, l, r)`. It will be represented as a pointer to a struct whose first field is a header encoding the head constructor `Node` (as well as other metadata), followed by the 64-bit representations of the four fields `c`, `v`, `l` and `r`.

In Ribbit, this generic way of representing any given ADT is available as a *generic representation*: writing `represented by caml` after any ADT definition will automatically generate the corresponding memory type.

2.6.2 Java Virtual Machine

Let us now focus on another uniform memory representation from a different ecosystem. The Java Virtual Machine is a platform supporting several languages, providing a common runtime framework with its own memory model. Like the OCaml runtime, it uses a rather uniform memory representation scheme to satisfy the demands of its garbage collector. Unlike OCaml, most JVM-based languages are heavily object-oriented: the basic memory management unit is an *object*, i.e., an instance of some class.

2.6.2.1 ADTs in JVM-based languages

The canonical JVM-based language is of course Java. Its type system is almost exclusively geared towards an object-oriented programming paradigm, with the majority of data being stored in class attributes. In this context, product types are easy to model as classes whose data attributes represent different fields, as seen in Exhibit 18.

(a) In Java

```
class Node {
  Color color;
  int value;
  RBT left;
  RBT right;
}
```

(b) In Scala

```
class Node(
  val color: Color,
  val value: int,
  val left: RBT,
  val right: RBT
)
```

(c) In Ribbit

```
struct Node {
  c: Color,
  v: u64,
  l: RBT,
  r: RBT
}
```

Exhibit 18: Product type for red-black tree nodes.

However, sum types are more delicate to model. Java provides an `enum` construct similar to C enums, which is sufficient for sum types whose constructors are all argument-less. For more complex sum types, a typical design pattern would be to create an abstract class from which every sum constructor inherits, adding its own data attributes. Data manipulation code may then use overloaded methods or a visitor pattern [1]. Exhibit 19 shows how Java enums and inheritance can be used to model red-black trees and their colors.

(a) In Java

```
enum Color { RED, BLACK; };

abstract class RBT {}

class EmptyRBT extends RBT {};
class NodeRBT extends RBT {
  Node node;
};
```

(b) In Scala

```
enum Color {
  case Red;
  case Black;
}
enum RBT {
  case Empty;
  case Node(Node);
}
```

(c) In Ribbit

```
enum Color { Red, Black }
enum RBT {
  Empty,
  Node(Node)
}
```

Exhibit 19: Sum types for red-black trees and their colors.

In addition to Java, the JVM also supports other languages which support richer types and different programming paradigms, most notably Scala. Specifically, Scala 3 introduces an enum feature [2] which a offers a more natural syntax for ADTs. We illustrate it on red-black tree types in Exhibits 18 and 19. Note that this feature is mostly syntactic sugar around advanced object-oriented features [3].

2.6.2.2 Internal representation of JVM objects

Now that we have described how to model ADTs in JVM-based languages, let us focus on their internal representation as JVM memory contents. All information about internal memory structures of the JVM was obtained by examining test Java programs (using OpenJDK 21 for x86-64 with compressed references enabled) with the JOL (Java Object Layout) tool [4], and is coherent with the analysis provided in https://shipilev.net/jvm/objects-inside-out.

Broadly speaking, every toplevel object is represented as a pointer to a block, similar to OCaml "boxed" values. The first 64 bits of this block are used by the *mark word*, which contains various metadata about the object itself, such as information used by the garbage collector. Next to the mark word, the 32-bit *class word* indicates which class this object is an instance of. After this 96-bit header, the block contains the memory representation of each field of the object. In order to minimize space usage while maintaining 64-bit alignment, the JVM may reorder fields so as to pack smaller fields within larger fields' alignment gaps (so-called "field packing"). Primitive fields such as integers are unboxed within their parent object's representation.

An interesting feature of many JVM implementations is the ability to *compress object references* [5]. This

[1] See for instance https://garciat.com/posts/java-adt for a practical example.
[2] https://github.com/lampepfl/dotty/issues/1970
[3] https://docs.scala-lang.org/scala3/reference/enums/desugarEnums.html
[4] https://openjdk.org/projects/code-tools/jol/
[5] https://shipilev.net/jvm/anatomy-quarks/23-compressed-references

optimization applies in situations where the host platform is a 64-bit machine, yet the size of the Java heap does not exceed 2^{32} bytes. In these situations, all pointers to Java objects (allocated on the heap) will necessarily be between 0 and $2^{32} - 1$. The 32 highest bits of a machine pointer to any Java object will therefore always be zero and can be discarded, enabling a significant reduction of memory usage.

The diagrams in Exhibit 20 show the memory representation of instances of two simple Java classes, illustrating both field packing and alignment gaps.

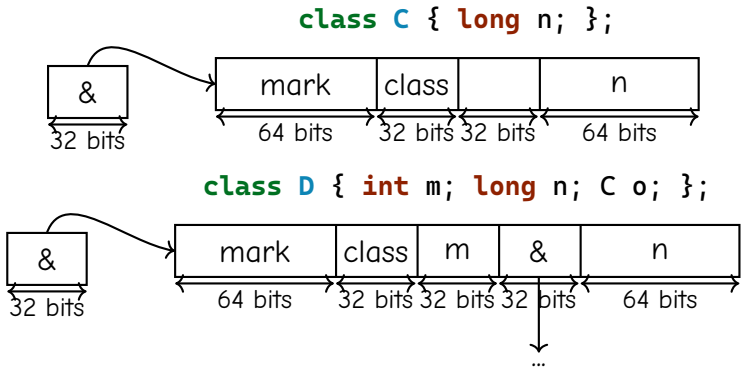


Exhibit 20: Objects in JVM memory.

In Ribbit, we would model an object's header and fields using ordinary structs and word types, including explicit padding. For compressed references, we must model the following semantics: "take only the 32 lowest bits of a 64-bit machine pointer, which contain all address bits". This is exactly the meaning of a Ribbit 32-bit pointer type (on a 64-bit platform), denoted `&<32>(...)`.

2.6.3 C

In contrast to OCaml and to JVM-based languages, the C programming language does not support high-level notions such as ADTs. Instead, C data types are directly expressed as their memory layout using constructs such as structs and unions.

This has a number of drawbacks. As C is a relatively low-level programming language, all data manipulation code must be written with its precise memory layout in mind, sometimes resulting in significantly obfuscated code, as we have seen in Section 2.1.4 with red-black trees in Linux. Furthermore, such low-level data types do not necessarily reflect intended type *semantics*, that is, the data structure the user actually had in mind, as opposed to its concrete implementation. As a result, none of the safety guarantees provided by ADTs are available: it is up to the user to write data manipulation code which is exhaustive, non-redundant and keep memory contents "well-typed" w.r.t. the intended high-level data structure.

Despite all these drawbacks, many performance-intensive applications are still written in C. Indeed, such programs often rely on manually specified, finely optimized memory layouts, whose specification requires total control over low-level details. This ability to precisely specify the desired memory representation of data is only afforded to users by C and a handful of other languages.

Here, we compare the language of C data types with Ribbit's memory types, highlighting their common features and key differences. All information presented in this section comes from the precise implementation of C data types. More precisely, our sources are the C17 standard and, for ABI-dependent aspects, the System V generic ABI v4.1 [6] and its processor supplement ABIs for x86_64 [7] and ARM32 [8] (these two ABIs do not differ on aspects which are relevant to us here).

Primitive data types C provides a variety of primitive data types (`int`, `char`, `long`, `double`, etc.), which are used both to actually encode a primitive of this type and as a way to get a raw word of this size. Ribbit separates actual primitive types (in practice, `uint`) from other usages of words:

[6] https://refspecs.linuxbase.org/elf/gabi41.pdf
[7] https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf
[8] https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst

uninitialized/unspecified contents `_<l>`, constants `(c)<l>`, composite `... with [o:l]:...`. In C, the intended contents of, say, an `int` are not encoded in its type.

Pointers C understands machine pointers to any given data type, as well as opaque pointers `void*`. In Ribbit, pointer types indicate both a specific pointee type and a width. If this width is different from that of a native pointer, it designates pointers whose higher bits have been extended or removed. Of course, this is only possible if shrinking the pointer in this way does not eat into address bits. For an example of such resized pointers, see JVM compressed references in Section 2.6.2.

Structs According to the C standard, the C compiler does *not* reorder struct fields, although it does automatically insert padding to meet alignment constraints. The contents of any padding are unspecified and may change when copying the struct. It is up to programmers to reorder fields to minimize the amount of space lost to alignment gaps; this technique is known as *struct packing* [9]. On the other hand, Ribbit does not reorder fields nor add any padding: the actual contents of a struct follow exactly its user-provided specification. This allows Ribbit to model arbitrarily weird encodings such as `{{(0)<7>, (1)<4>}}`, which will indeed be 11-bit wide.

Enums can be seen as very simple sum types with only unit constructors, each represented as a value of a `char` or any suitable signed or unsigned integer type – the actual primitive type used to store enum values is implementation-defined. Note that there is no guarantee that the concrete contents of any value of a given enum type actually represent one of its inhabitants (i.e., any integer of the right width can be casted to a (possibly non-sensical) enum value). In Ribbit, C-like enums correspond to a sum type with only unit constructors represented as a split in which each branch represents one constructor as a constant word. The predefined representation `represented by C`, which only works on such ADTs, will automatically find the minimal width necessary to encode all constructors (not necessarily a power of 2, for instance we encode RISC-V registers on 5 bits in Section 2.5) and generate such a split.

Unions can be seen as degenerate splits without an explicit discriminant. Similar to structs, the C compiler will automatically add padding at the end of smaller variants to meet the alignment of the largest variant. A possible representation of a sum type in C is a *tagged union*, i.e., a struct which aggregates an explicit enumerated *tag* with its union *payload*. This pattern is captured and generalized in a safe way by Ribbit split types.

Bit-fields are an alternative way to specify members of a struct or an union. They are rarely used by C programmers in practice. C structs containing bit-fields resemble "packed" Ribbit-like structs, but also rely on a notion of "storage units" (usually machine words). Whether bit-fields can straddle storage unit boundaries is implementation-defined; as such, using bit-fields to specify precise layout details is rather unreliable.

2.6.4 Rust

Rust is probably the most promising mainstream language when it comes to combining ADTs with optimized memory layouts. Like Ribbit, Rust aims to offer as much low-level control as C while still providing nice and safe abstractions such as ADTs and pattern matching. The syntax of Ribbit for ADTs and pattern matching is heavily inspired by (and thus basically identical to) that of Rust. Here, we focus on the memory representation of ADT values in Rust as described in its documentation (*The Rust Reference* 2023; *The Rustonomicon* 2023), and on the (limited) ways in which Rust programmers can customize this representation.

Every ADT defined in Rust can carry an annotation of the form `#[repr(...)]` which specifies one of four possible memory representation schemes, optionally modified with `packed` or `align` attributes to customize object alignment. We describe each of them below.

C representation As its name implies, the C representation aims to closely follow the memory layout defined by the C standard, which we described in Section 2.6.3. Rust enums (a.k.a. sum types), which have no native C equivalent, are represented as "tagged unions". It is useful for interfacing

[9] http://www.catb.org/esr/structure-packing

with C (and other cooperative languages) through Rust's Foreign Function Interface, but also for providing a mostly predictable representation of values. As we will see, the main other available representation (Rust representation) is highly variable and purposefully unspecified.

Primitive representation The primitive representations (for instance, `#[repr(u8)]`) only apply to sum types. Essentially, they allow the user to choose which primitive type backs the representation of a C-like enum. For instance, `#[repr(u8)]` will guarantee that the given enum will be represented as an unsigned 8-bit integer. For sum types with non-unit constructors, the given primitive type instead specifies the type of the "tag" within the "tagged union" representation. This representation is easily specified in Ribbit, similarly to the C language's enums.

Transparent representation The transparent representation applies to product types with a single field, and to sum types with only one constructor which itself contains a single field. It represents values of such a type as their *unboxed* field, similar to the `unboxed` OCaml attribute [10]. Again, this representation is easy to express as a Ribbit memory type.

Rust representation Unlike the three previous representations, the Rust reference states that the (default) Rust representation makes no data layout guarantees, except those required for soundness (e.g., the fields of a struct do not overlap each other). According to the Rustonomicon, struct fields may be reordered by the compiler, and padding is inserted as needed to meet alignment constraints. On the other hand, the memory layout of enums is purposefully left unspecified. Indeed, the Rust compiler reserves itself the right to apply arbitrary data layout optimizations, without any explicit input from the programmer. A well-known example is the popular *niche* optimization, which takes advantage of unused values to represent extra sum constructors. For instance, consider an option type wrapping pointer values: `Option<Box<...>>`. Since 0 is not a valid address, the Rust compiler will use this value to represent the `None` value, whereas values of the form `Some(p)` will be represented as their unboxed pointer value `p`. Doing so saves space – no extra space is used for the tag – and improves performance by removing any overhead associated with the `Some` wrapper. This particular optimization, although expressible in Ribbit syntax, is not currently handled in its formal version nor by its implementation.

2.7 Limits of Ribbit: WebKit-like NaN-boxing

As the final section of our Memory Zoo, we describe a representation that models the memory layout used in the JavaScriptCore engine (built into WebKit) to encode JavaScript values on 64-bit platforms, which uses an optimization dubbed *NaN-boxing*. In doing so, we will expose some of Ribbit's limitations.

Our description is based off the implementation of *WebKit NaN-boxing* (2023). According to the *ECMAScript Language Types* (2023), JavaScript values consist of:

- four constants: *undefined*, *null* and the boolean values *true* and *false*;
- *numbers*, which consist of 32-bit integers and double-precision (64-bit) floating-point numbers;
- arbitrary-precision integers, character strings, symbols and objects, which we collectively refer to as *cells*. Every cell value is represented behind a pointer; here, we ignore other representation details.

We define the type of Javascript values in Ribbit as the `JSVal` ADT shown in Exhibit 21, Line 5. The memory type that represents `JSVal` values according to the layout used in JavaScriptCore is based on the double-precision binary encoding defined by the IEEE 754 standard and takes advantage of unused *NaN* values to represent all JavaScript values as 64-bit words.

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

[10] https://ocaml.org/manual/5.2/attributes.html

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

The JavaScriptCore implementation also takes advantage of the fact that no current x86-64 implementation uses more than 2^{48} bytes of virtual address space, that is, 48 bits are sufficient to store any machine pointer. In order to keep pointer dereferencing unencumbered by extra decoding operations, pointers are assigned the range from 0 inclusive to 2^{48} exclusive. The four constants (`Undef`, `Null`, `True` and `False`) are mapped to values in this range, using the fixed invalid pointer values `0xa`, `0x2`, `0x7` and `0x6` respectively.

We model this encoding as a Ribbit memory type in Exhibit 21. As seen on line 6, three main categories of values are distinguished based on their 16 high bits. Lines 8 to 14 cover all non-numeric values (constants and cells).

```
1  type Cell = i256 repr as i256;
2
3  enum Num { Int(i32), Double(f64) }
4
5  enum JSVal { Undef, Null, Bool(Bool), Num(Num), CellRef(Cell) }
6  represented as split .[48:16] {
7    | 0 from (CellRef(_) | Undef | Null | Bool) =>
8      split .[0:48] {
9      | 6 from Bool(False) => _<64> with [0:48] : (6)<48>
10     | 7 from Bool(True) => _<64> with [0:48] : (7)<48>
11     | 10 from Undef => _<64> with [0:48] : (10)<48>
12     | 2 from Null => _<64> with [0:48] : (2)<48>
13     | _ from CellRef(_) => _<64> with [0:48] : &<48>(.CellRef as Cell)
14     } with .[48:16] : (0)<16>
15   | 0xfffe from Num(Int(_)) =>
16     _<64> with [0:32] : (.Num.Int as i32) with [32:16] : (0)<16> with [48:16] : (0xfffe)<16>
17   | _ from Num(Double(_)) =>
18     _<64> with [0:48] : ((u48)(.Num.Double.[0:48]) as u48)
19     with [48:16] : ((u16)(.Num.Double.[48:16]) + 1 as u16)
20 }
```

Exhibit 21: Javascript values and memory layout using NaN-boxing, in Ribbit syntax.

32-bit integer values are assigned the range from `0xfffe000000000000` to `0xfffe0000ffffffff` inclusive, and are distinguished by their 16 higher bits (`0xfffe`), as seen on line 15. We then use the standard 32-bit integer encoding on the 32 lowest bits.

Finally, the range of double-precision numbers is offset by 2^{49}, making its exclusive upper bound `0xfffa000000000000`, so that it lies outside both pointer and integer value ranges. It corresponds to the default split branch line 17. As the binary encoding of 2^{49} contains only zeroes as its 48 lowest bits, we only need to add this arithmetic offset to the 16 highest bits of the `Num(Double(_))` representation. On Line 19, we take the 16 highest bits of the `f64` representation of `.Num.Double`, cast them to an unsigned 16-bit integer `u16` and finally add 1 (which is 2^{49} shifted right by 48 bits) to obtain the desired value.

While our syntax is expressive enough to describe the `JSVal` memory layout, some of its features are not supported by Ribbit beyond its syntax. They fall outside of the scope of the Ribbitulus (formalization of the Ribbit language described in Chapter 3) and are not handled by the ribbit compilation algorithms (which we describe in Chapters 4 and 5). These unsupported features are:

- Primitive types beyond integers – for our example, double-precision floats. For simplicity, the current formalization and implementation of Ribbit only support integer primitives.
- Complex primitive encodings in which reversible operations are applied to raw primitive values. In our example, we use bitcasts (from portions of an `f64` to unsigned integers `u48` and `u16`) and

constant integer addition (`+ 1`). Currently, Ribbit only supports the extraction of specific bit ranges from integer values (e.g., `.Num.Double.[0:48]`).

- Wildcard discriminant values in splits, indicating that a given split branch applies to memory values whose contents at the discriminant position are *not* matched by other branches. The current formalization of Ribbit requires split branches' memory types to explicitly contain their discriminant value as a constant at the appropriate position.

- Awareness of low-level details regarding the contents of machine pointers and primitive encodings. Currently, pointers and primitives are treated as "opaque" values in memory whose contents are completely unpredictable. However, depending on the considered system and architecture, some information is in fact available. For instance, the Rust compiler exploits the fact that valid pointers are never zero to encode pointer options on the same width as a pointer, by using the value `0` to represent `None`. Such layout optimizations are known as *niches*. In our example, checking that the split type is indeed valid would require Ribbit to "know" that `6`, `7`, `10` and `2` are not valid addresses, and that the 16 upper bits of a double-precision float are always between 0 inclusive and `0xfff8` exclusive.

The first three features would probably be reasonably easy to fit into the existing Ribbit formalism and implementation. However, properly modeling niches would most likely require a formalization of the precise details of numeric encodings and of memory address allocation for a given system and architecture. Ideally, such a formalization could then be integrated into Ribbit to allow for target-specific layout optimizations; however, it is outside the scope of this thesis.

2.8 Conclusion

In this chapter, we have explored ADTs and their memory representations in a wide variety of contexts. We used this opportunity to introduce our Ribbit language, which combines high-level, safe abstractions – namely ADTs and pattern matching – with a memory description language allowing for precise specification of data layout by the user.

In the next chapter, we will go beyond this surface-level description and fully formalize the Ribbit language, including its safety properties.

Chapter 3

The Ribbitulus

This chapter presents a formalization of the ribbit language presented in Chapter 2 dubbed the *Ribbitulus*. In Section 3.1, we define a formal syntax which captures both high-level and memory elements of our language, along with tools to manipulate these syntactic constructs. Section 3.2 introduces various typing and validity judgments which characterize well-formed ribbit programs. Section 3.3 defines a two-tiered formal semantics for the ribbit language, with high-level and memory-level evaluation judgments. Finally, we state and prove the soundness of our semantics and establish an equivalence between high-level and memory-level behavior of ribbit programs in Section 3.4. Figure 3.34 provides an index of notations introduced in this chapter.

3.1 Syntax

We first formalize our input language. As in Chapter 2, we present a two-tiered view: *high-level* types used for programming and following a common presentation of ADTs, and *memory layout* specifications detailing how to represent them in memory. We also detail the grammar of programs we consider for our formal semantics and compilation algorithms.

3.1.1 High-level language

The high-level syntax of the Ribbitulus consists of Algebraic Data Types and their values, as well as other objects, all sharing common syntactical constructs. In definitions that apply to different kinds of objects, for instance to types, values and patterns, we will use the meta-variable θ to denote any such object.

3.1.1.1 Types

$$
\begin{array}{rll}
\tau \in \mathit{Types} ::= & \mathrm{t} \in \mathit{TyVars} & \text{(type variable)} \\
\mid & \mathrm{I}_\ell & \text{(ℓ-bit wide unsigned integer primitive type)} \\
\mid & \langle \tau_0, \ldots, \tau_{n-1} \rangle & \text{(tuple/product type with n fields)} \\
\mid & \mathrm{K}_0(\tau_0)\text{`|'}\ldots\text{`|'}\mathrm{K}_{n-1}(\tau_{n-1}) & \text{(enum/sum type with n constructors)} \\
\Delta : & \mathit{TyVars} \to \mathit{Types} & \text{(type variable environment)}
\end{array}
$$

Figure 3.1: Algebraic Data Types in the high-level language.

Our source language features simple (monomorphic, immutable) algebraic data types whose grammar is presented in Fig. 3.1. We denote types using τ and type variables with t. We restrict primitive types to unsigned integers of a given width ℓ (in bits), denoted I_ℓ. We denote all tuples with angle brackets,

for instance $\langle I_{64}, I_{64}\rangle$ for the type of pairs of 64-bit integers. Constructors of sums are marked with a capital letter, for instance "Some(t) | None" is an option type. In examples, we use K as a shortcut for $K(\langle\rangle)$. In addition, we use Δ to denote type name environments, i.e., maps from type variables t to types τ, which we use to model recursive types.

Example 3.1 (Running example: lists)**.** Consider the type $\tau_{list} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{list}\rangle)$ in the type variable environment $\Delta_{list} = \{t_{list} \mapsto \tau_{list}\}$, which formalizes the `List` ADT from Section 2.4. In the remainder of this chapter, we will regularly refer to τ_{list} and Δ_{list} from this example to illustrate various notions. △

3.1.1.2 Patterns and Specialization

Patterns, denoted p and defined in Fig. 3.2, describe the "shape" of a value with tuples, constructors, primitive constants and wildcards denoted _. P denotes a set of patterns.

$$
\begin{aligned}
p \in \mathit{Patterns} ::= {} & \_ && \text{(wildcard)}\\
\mid {} & c && \text{(primitive constant)}\\
\mid {} & \langle p_0, \ldots, p_{n-1}\rangle && \text{(tuple)}\\
\mid {} & K(p) && \text{(constructor)}
\end{aligned}
$$

Figure 3.2: Patterns.

For instance, given the type None | Some(τ), the pattern Some(_) matches values whose head constructor is Some. We will formally define the semantics of pattern matching in Section 3.3.1.

The *specialization* operation, denoted τ / p and defined in Fig. 3.3, restricts the type τ to values matching the pattern p by filtering out constructors that do not appear in p. For instance, (None | Some(τ)) $/$ Some(_) = Some(τ).

$$
\tau/\_ = \tau \qquad I_\ell / c = I_\ell \qquad \langle \tau_0, \ldots, \tau_{n-1}\rangle / \langle p_0, \ldots, p_{n-1}\rangle = \left\langle \tau_0/p_0, \ldots, \tau_{n-1}/p_{n-1}\right\rangle
$$

$$
K_0(\tau_0) \mid \cdots \mid K_{n-1}(\tau_{n-1}) / K_i(p) = K_i(\tau_i / p)
$$

Figure 3.3: Specialization of a type according to a pattern.

Given two patterns p and p′, their intersection $p \sqcap p'$ captures values that match both p and p′, as defined in Fig. 3.4. If $p \sqcap p'$ is undefined, then p and p′ are said to be *incompatible*. For instance, $\_ \sqcap$ Some(_) = Some(_), and None is incompatible with Some(_).

$$
\_ \sqcap p = p \sqcap \_ = p \qquad c \sqcap c = c \qquad \langle p_0, \ldots, p_{n-1}\rangle \sqcap \left\langle p'_0, \ldots, p'_{n-1}\right\rangle = \left\langle p_0 \sqcap p'_0, \ldots, p_{n-1} \sqcap p'_{n-1}\right\rangle
$$

$$
K(p) \sqcap K(p') = K(p \sqcap p')
$$

Figure 3.4: Intersection of two patterns.

3.1.1.3 Paths and Focusing

Paths, denoted π and defined in Fig. 3.5, indicate a position within a type, pattern or valuexpression (defined later, in Fig. 3.8). Given a path π and $\theta \in \mathit{Types} \cup \mathit{ValuExprs} \cup \mathit{Patterns}$, we denote **focus** (π, θ) the subterm at position π within θ. In addition to field and constructor accesses, which focus within a product or sum type respectively, a path may contain a *bit range*, denoted r, as its last operation to focus on individual bits within integers. More precisely, $[o : \ell]$ designates ℓ contiguous bits from offset

o inclusive. For instance, in the type $\tau = \text{Some}(\langle I_8, t\rangle) \mid \text{None}$, the least significant bit of the I_8 is located at $\pi = .\text{Some}.0.[0:1]$ and we have **focus** $(\pi, \tau) = I_1$. Focusing also unfolds type variables as necessary. The full focusing operation is defined in Fig. 3.6.

$$
\begin{array}{rl}
r ::= [o : \ell] & \text{(bit range)} \\
\pi \in \mathit{Paths} ::= \varepsilon & \text{(empty path)} \\
\mid r & \text{(integer bit range extraction)} \\
\mid .i.\pi & \text{(tuple field access)} \\
\mid .K.\pi & \text{(constructor access)}
\end{array}
$$

Figure 3.5: Paths indicating a position within a high-level term.

focus{

$$
\begin{array}{lll}
\varepsilon & , \theta & \longrightarrow \theta \\
.[o:\ell].\pi & , I_{\ell_0} & \longrightarrow \textbf{focus}\,(\pi, I_\ell) \text{ when } o + \ell < \ell_0 \\
.[o:\ell].\pi & , c & \longrightarrow \textbf{focus}\,(\pi, c') \text{ where } c' = (c >> o) \wedge (2^\ell - 1) \\
.i.\pi & , \langle \theta_0, \ldots, \theta_{n-1}\rangle & \longrightarrow \textbf{focus}\,(\pi, \theta_i) \\
.K_i.\pi & , K_0(\tau_0) \mid \cdots \mid K_{n-1}(\tau_{n-1}) & \longrightarrow \textbf{focus}\,(\pi, \tau_i) \\
.K.\pi & , K(\theta) & \longrightarrow \textbf{focus}\,(\pi, \theta) \\
\pi & , t & \longrightarrow \textbf{focus}\,(\pi, \Delta(t)) \\
\pi & , x.\pi_0 & \longrightarrow x.(\pi_0.\pi) \\
\pi & , \_ & \longrightarrow \_
\end{array}
$$

}

Figure 3.6: Focus on the subterm at position π within $\theta \in \mathit{Types} \cup \mathit{ValuExprs} \cup \mathit{Patterns}$, using the type variable environment Δ to dereference type variables. $>>$ denotes the bitwise logical right shift operation and $\wedge$ the bitwise logical and operation; their combination allows us to extract the desired range of bits from an integer c.

Similar to specialization by a pattern, τ / π defined in Fig. 3.7 restricts τ to values v for which **focus** (π, v) is defined. Note that unlike focusing, this does not return a subterm of τ. For instance, $(\text{None} \mid \text{Some}(\tau)) \,/\, .\text{Some} = \text{Some}(\tau)$.

$$
\tau / \varepsilon = \tau \qquad \frac{o + \ell' \leqslant \ell}{I_\ell \,/\, .[o:\ell'] = I_\ell} \qquad \langle \tau_0, \ldots, \tau_{n-1}\rangle \,/\, .i.\pi = \left\langle \tau_0, \ldots, \tau_{i-1}, \tau_i / \pi, \tau_{i+1}, \ldots, \tau_{n-1}\right\rangle
$$

$$
K_0(\tau_0) \mid \cdots \mid K_{n-1}(\tau_{n-1}) / .K_i.\pi = K_i(\tau_i / \pi)
$$

Figure 3.7: Specialization of a type according to a path.

3.1.1.4 Source programs

ADT valuexpressions

$$
\begin{aligned}
u \in \mathit{ValuExprs} ::= {} & x.\pi && \text{(variable subterm accessor)} \\
\mid {} & c \in \mathbb{N} && \text{(unsigned integer constant)} \\
\mid {} & \langle u_0, \ldots, u_{n-1} \rangle && \text{(tuple)} \\
\mid {} & K(u) && \text{(constructor)} \\
v \in \mathit{Values} ::= {} & c \mid \langle v_0, \ldots, v_{n-1} \rangle \mid K(v)
\end{aligned}
$$

Full expressions

$$
\begin{aligned}
e \in \mathit{Exprs} ::= {} & (u : \tau \text{ as } \widehat{\tau}) && \text{(pivot expression)} \\
\mid {} & \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = e \ \mathsf{in}\ e' && \text{(value binding)} \\
\mid {} & f(x) && \text{(function application)} \\
\mid {} & \mathsf{match}(x)\{p_0 \to e_0 \ \ldots \ p_{n-1} \to e_{n-1}\} && \text{(pattern matching)}
\end{aligned}
$$

Figure 3.8: Source expressions and values.

We formalize input programs as simplified expressions (shown in Fig. 3.8) where every expression is let-bound, akin to A-normal form (Sabry and Felleisen 1993). *Full expressions* include function applications $f(x)$, pattern matching, and let-bindings. We assume that all function definitions have been processed into an environment denoted Σ binding each function symbol f to a term of the form $\lambda x.e$. Pattern matching constructs, as introduced in Section 2.1.1, consist of rules which filter value shapes with a pattern on their left-hand side and return the expression on their right-hand side. Let-bindings are annotated with both a type τ and a *memory type* $\widehat{\tau}$, which corresponds to the memory layout specification part of our language and will be explained in Section 3.1. Finally, *pivot* expressions, of the form $(u : \tau \text{ as } \widehat{\tau})$, describe a concrete value of type τ whose representation in memory should follow the memory type $\widehat{\tau}$. The type and memory type in pivot expressions may be omitted when they are immediate from context (for instance, $\mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = u \ \mathsf{in}\ e$ is a syntactic shorthand for $\mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = (u : \tau \text{ as } \widehat{\tau}) \ \mathsf{in}\ e$). *Valuexpressions*, denoted u, have a syntax reminiscent of types, consisting of tuples and constructors, along with integer constants $c \in \mathbb{N}$. They also introduce *accessors* of the form $x.\pi$, representing the subterm located at π within the value bound to x. For simplicity, we do not support recursive values (even though types might be recursive). ADT values are the subset of valuexpressions that do not contain any accessors.

Both valuexpressions and patterns can be focused on at a specific position, using the same focusing operation defined in Fig. 3.6 as types. For instance, **focus** $(.K.0, K(x)) = x.0$.

Example 3.2 (Full expression with lists)**.** Given a memory type $\widehat{\tau}$, the following expression:

$$
\mathsf{let}\ x : \tau_{\text{list}} \text{ as } \widehat{\tau} = \text{Cons}(\langle 42, \text{Nil} \rangle) \ \mathsf{in}\ \mathsf{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\langle \_, \_ \rangle) & \to (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\}
$$

1. builds the value $\text{Cons}(\langle 42, \text{Nil} \rangle)$ of type τ_{list}, assigns the memory layout $\widehat{\tau}$ to it and binds it to x;
2. matches this value against two patterns;
3. returns a value of type I_{32} (represented as a standard 32-bit integer in memory), either the constant 0 or the subterm at position .Cons.0 in x.

△

3.1.2 Layout specification with memory types

We now formalize the second part of our language, which consists of user-specified memory layouts in the form of *memory types*. As seen in Fig. 3.8, each high-level expression is associated with a *memory*

type, denoted $\widehat{\tau}$, which specifies its memory layout. As a convention, all memory objects are given a hat. For instance memory types, denoted $\widehat{\tau}$ and defined in Fig. 3.9, describe how values of a given ADT τ will be represented in memory. At this representation level, we are bit-precise, yet abstract away some architecture-dependent details such as endianness and machine pointer size and address alignment. Let us first ignore the "split" alternation, and focus on the rest of the grammar. We extend the codomain of Δ so that it now contains both high-level and memory-level type variable bindings – that is, it maps each type variable to either a high-level type or a memory type.

$$
\begin{aligned}
\widehat{\tau} \in \widehat{\mathit{Types}} ::= {} & t \in \mathit{TyVars} && \text{(type variable)}\\
\mid {} & \__\ell && (\ell\text{-bit wide word of unspecified contents})\\
\mid {} & (c)_\ell && (\ell\text{-bit wide immediate encoding the constant } c)\\
\mid {} & \&_\ell\,(\widehat{\tau}) && (\ell\text{-bit wide pointer to a } \widehat{\tau} \text{ value})\\
\mid {} & \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i && (\text{composite word type with } n \text{ extra values stored in unused bits of } \widehat{\tau})\\
\mid {} & \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} && (n\text{-field struct})\\
\mid {} & I_\ell && (\ell\text{-bit wide unsigned integer encoding})\\
\mid {} & (\pi \text{ as } \widehat{\tau}) && (\text{fragment representing the subterm at position } \pi \text{ as } \widehat{\tau})\\
\mid {} & \mathrm{split}\,(\widehat{\pi}_0, \ldots, \widehat{\pi}_{N-1})\left\{ c_{i,0}, \ldots, c_{i,N-1} \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n \right\} && \\
& && (\text{split with N discriminant locations and } n \text{ branches of provenances } P_i)\\
\Delta : \mathit{TyVars} \to {} & \mathit{Types} \cup \widehat{\mathit{Types}} && (\text{type variable environment})
\end{aligned}
$$

Figure 3.9: Memory types – the hat on $\widehat{\tau}$ distinguishes them from high-level types τ.

Fragments, denoted $(\pi \text{ as } \widehat{\tau})$, indicate that the subterm at the position π *in the high-level type* will be represented by the memory type $\widehat{\tau}$. As a special case, the "atomic" integer type I_ℓ encodes an integer value using the standard unsigned integer encoding on ℓ bits – it is equivalent to the fragment $(\varepsilon \text{ as } I_\ell)$. The contents of a memory word of a fixed width ℓ (in bits) may be left unspecified with $\__\ell$, set to a constant c with $(c)_\ell$, or filled with the address of a value of another memory type $\widehat{\tau}$ with the pointer type $\&_\ell\,(\widehat{\tau})$. Structure types are denoted by $\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$.

Example 3.3 (Memory type description)**.** Let us consider the high-level types I_{64} and $\tau_{\text{tup}} = \langle I_{32}, \tau, I_8 \rangle$ where τ is an arbitrary high-level type associated with some memory type $\widehat{\tau}$. We have a straightforward memory encoding for I_{64}: $\widehat{\tau}_{\text{int}} = I_{64}$ which encodes a 64-bit integer as "itself". For τ_{tup}, we choose to represent the tuple with a "struct", but we can choose the order of the components: first, we store the I_{32} (position .0 in the high-level type), then the I_8 (position .2), then some 24-bit padding (with zeroes), then a 64-bit pointer to the τ value represented as $\widehat{\tau}$. We group the I_8,I_{32} and I_{24} padding bits together and get $\widehat{\tau}_{\text{tup}} = \{\!\{(.0 \text{ as } I_{32}), (.2 \text{ as } I_8), (0)_{24}, \&_{64}\,((.1 \text{ as } \widehat{\tau}))\}\!\}$. △

Some of the memory types we have seen so far contain bits whose contents are unspecified. This is the case for types of the form $\__\ell$, but also for pointers, depending on architecture-specific characteristics. For instance, if $\widehat{\tau}$ corresponds to an ℓ-byte aligned structure, then the ℓ lower bits of the address of any such structure – that is, of any $\&_\ell\,(\widehat{\tau})$ value – are always zero. They can therefore be used to store extra information – this technique, used in C programs such as parts of the Linux kernel described in Section 2.1.4, is known as *bit-stealing*. On some systems, the sign bit of user-space pointers may be reclaimed in a similar way [1], as well as bits corresponding to unused virtual address space [2].

We capture this notion of extra information stored in the unused bits of a word or pointer with *composite words*. Given a memory type $\widehat{\tau}$ and n pairs $(r_i, \widehat{\tau}_i)$ each consisting of a bit range and a memory type, $\widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i$ denotes the type of values consisting of a "base" value of type $\widehat{\tau}$ where, for each i, the bits in r_i are used to store another value of type $\widehat{\tau}_i$. We may also denote composite words *in extenso* with $\widehat{\tau} \ltimes r_0 : \widehat{\tau}_0 \ltimes \cdots \ltimes r_{n-1} : \widehat{\tau}_{n-1}$ (observe the priority of ':' on '$\ltimes$'). For instance, as we will see later on,

[1] See for instance https://docs.rs/ointers/latest/ointers/.
[2] This is done in the NaN-boxing layout from Section 2.7.

the memory representation of the Nil list constructor according to the simple linked-list memory layout is $\_{}_{64} \ltimes [0:1] : (1)_1$, which denotes an uninitialized 64-bit word whose lowest bit is set to 1.

The memory type constructs described so far are sufficient to specify the layout of virtually any ADT combining integer and product types. However, none of these constructs are able to capture the notion of different layouts for distinct constructors of a sum type. To this end, we introduce the *split* construct. Splits model constraints of the form "if the value in memory at a given position is equal to some constant c, then we use the following memory type". In $\text{split}\,(\widehat{\pi}_1, \ldots, \widehat{\pi}_N)\,\{B\}$, the $\widehat{\pi}_j$ are N *discriminant positions* and B is a set of *branches* of the form $c_{i,1}, \ldots, c_{i,N}$ *from* $P_i \Rightarrow \widehat{\tau}_i$ where P_i is a set of constant-free patterns dubbed the *provenances* of the branch. It indicates that, if the value at each position $\widehat{\pi}_j$ in memory is $c_{i,j}$, then it represents a value matched by a pattern in the set P_i using the memory type $\widehat{\tau}_i$. If P_i is a singleton, we may write its single element without the surrounding curly braces.

Discriminant positions follow the grammar of *memory paths* defined in Fig. 3.10. Memory path operations include pointer dereferencing and struct field accesses, as well as operations that manipulate words on a smaller scale: extraction of ℓ bits from offset o, denoted $.[o:\ell]$, and bitwise "and" with an arbitrary (appropriately sized) bit mask $\mathfrak{m}$, denoted $.\mathfrak{m}$.

$$
\begin{aligned}
\mathfrak{m} &::= \varepsilon \mid 0.\mathfrak{m} \mid 1.\mathfrak{m} && \text{(bit mask)}\\
\widehat{\pi} \in \widehat{\mathit{Paths}} &::= \varepsilon && \text{(empty path)}\\
&\mid .r.\widehat{\pi} && \text{(bit range extraction)}\\
&\mid .\mathfrak{m}.\widehat{\pi} && \text{(bitwise and)}\\
&\mid .*.\widehat{\pi} && \text{(dereference)}\\
&\mid .i.\widehat{\pi} && \text{(struct field access)}
\end{aligned}
$$

Figure 3.10: Memory paths. Sets of memory paths are denoted $\widehat{\Pi}$.

Bit masks are sequences of bits written with the most significant bit on the left; we write 0^ℓ (resp. 1^ℓ) for ℓ contiguous zeroes (resp. ones), and $\mathfrak{m}_1 \ldots . \mathfrak{m}_n$ for the concatenation of n bit masks. $|\mathfrak{m}|$ denotes the total number of bits in $\mathfrak{m}$. In practice, we will almost always use bitwise ands to "mask off" a given bit range from a composite word; to this end, $\neg[o:\ell]$ denotes the bit mask whose total width is evident from context, whose bits o inclusive to $o+\ell$ exclusive are set to zero and whose other bits are set to one. For instance, given the memory type $\widehat{\tau} = \_{}_{64} \ltimes [0:32] : (42)_{32} \ltimes [32:32] : (0)_{32}$, the bit mask $\neg[0:32]$ is implied to be 64 bits wide; when applied to $\widehat{\tau}$, it designates the type $\_{}_{64} \ltimes [32:32] : (0)_{32}$ – that is, the specification applied to the 32 lowest bits in $\widehat{\tau}$ has been masked off.

Example 3.4 (Splits in lists)**.** Consider the high-level type τ_{list} in the type variable environment Δ_{list} from Example 3.1. We model the two memory layouts described in Exhibit 13 with two different memory types. We first encode the naive (modulo pointer tagging) layout `List` as the following memory type:

$$
\widehat{\tau}_c = \text{split}\,(.[0:1])\left\{
\begin{array}{lll}
1 \text{ from } \text{Nil} & \Rightarrow & \_{}_{64} \ltimes [0:1] : (1)_1 \\
0 \text{ from } \text{Cons}(\_) & \Rightarrow & \&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1 as } t_c)\}\!\}) \ltimes [0:1] : (0)_1
\end{array}
\right\}
$$

As this memory layout is fundamentally recursive, we add a new binding to our type variable environment:

$$\Delta_{\text{list}} = \{t_{\text{list}} \mapsto \tau_{\text{list}}, t_c \mapsto \widehat{\tau}_c\}$$

Since there are two possible representations depending on the head constructor, we use a "split" type with two branches. In the first branch, the single pattern Nil indicates that it represents the Nil high-level value. Its memory type is $\_{}_{64} \ltimes [0:1] : (1)_1$: a 64-bit word whose lowest bit is set to 1. In the second branch, the provenance Cons(_) encompasses all other τ_{list} values and represents them using the memory type $\&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1 as } \widehat{\tau}_c)\}\!\}) \ltimes [0:1] : (0)_1$, which is a 64-bit wide pointer whose lowest bit (address alignment bit) is set to 0, pointing to a struct encoding the first element of the list as a 32-bit integer, followed by the list of remaining elements, itself represented as $\widehat{\tau}_c$. Finally, the split discriminant $.[0:1]$ indicates how to tell these two cases apart: by looking at the lowest bit, which

is 1 in the Nil case and 0 in the Cons case, as enforced in both memory types with a composite word specification ($\ltimes[0:1]$).

Another possible choice of representation for the same ADT is the "packed" layout `PairList`, which we model with the following memory type:

$$
\begin{array}{l}
\widehat{\tau}_p = \mathrm{split}\,(.[0:2])\,\{ \\
\quad 0 \;\text{from}\; \mathrm{Nil} \qquad\qquad\qquad\quad \Rightarrow\; \_{64} \ltimes [0:2] : (0)_2 \\
\quad 1 \;\text{from}\; \mathrm{Cons}(\langle \_, \mathrm{Nil} \rangle) \quad \Rightarrow\; \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } I_{32}) \\
\quad 2 \;\text{from}\; \mathrm{Cons}(\langle \_, \mathrm{Cons}(\_) \rangle) \;\Rightarrow\; \&_{64}\left(\{\!\{(.\mathrm{Cons}.0 \text{ as } I_{32}), (.\mathrm{Cons}.1.\mathrm{Cons}.0 \text{ as } I_{32}), (.\mathrm{Cons}.1.\mathrm{Cons}.1 \text{ as } t_p)\}\!\}\right) \\
\qquad\qquad\qquad\qquad\qquad\quad \ltimes [0:2] : (2)_2 \\
\}
\end{array}
$$

Again, we add a new binding to our type variable environment for this recursive memory type:

$$\Delta_{\mathrm{list}} = \{t_{\mathrm{list}} \mapsto \tau_{\mathrm{list}}, t_c \mapsto \widehat{\tau}_c, t_p \mapsto \widehat{\tau}_p\}$$

We fit up to two elements per level of indirection, using a three-branch split whose first two branches represent empty and singleton lists similarly to the previous layout $\widehat{\tau}_c$, and whose last branch inlines the two first elements of longer lists into a struct. △

The size of a memory type $\widehat{\tau}$, denoted $|\widehat{\tau}|$ and defined in Fig. 3.11, has a rather straightforward definition. Note that we only consider memory types whose size can be computed; in practice, this means that recursive types must always introduce some form of indirection. For instance, in the type variable environment $\{t \mapsto \widehat{\tau}\}$, this criterion forbids $\widehat{\tau} = \{\!\{\__\ell, t\}\!\}$ but allows $\widehat{\tau} = \{\!\{\__\ell, \&_\ell(t)\}\!\}$.

$$
|t| = |\Delta(t)| \qquad |\__\ell| = \ell \qquad |(c)_\ell| = \ell \qquad |\&_\ell(\widehat{\tau})| = \ell \qquad \left|\widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i\right| = |\widehat{\tau}|
$$

$$
|\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}| = |\widehat{\tau}_0| + \cdots + |\widehat{\tau}_{n-1}| \qquad |I_\ell| = \ell \qquad |(\pi \text{ as } \widehat{\tau})| = |\widehat{\tau}|
$$

$$
\left|\mathrm{split}\,(\widehat{\pi}_1, \ldots, \widehat{\pi}_N)\left\{ c_{i,1}, \ldots, c_{i,N} \;\text{from}\; P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 1 \leqslant i \leqslant n \right\}\right| = \max_{1 \leqslant i \leqslant n} |\widehat{\tau}_i|
$$

Figure 3.11: Size of a memory type $\widehat{\tau}$ in the type variable environment Δ.

We formalize the notion of "memory contents at a given position $\widehat{\pi}$ in a memory type $\widehat{\tau}$" with a *memory focusing* operation, denoted $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau})$ and defined in Fig. 3.12. It is similar to focusing on high-level terms, with the caveat that "focusing below a bitmask" is akin to performing a bitwise "and", as explained above.

$$
\begin{array}{lll}
\widehat{\mathbf{focus}}_\Delta\{ & & \\
\quad \varepsilon & , \widehat{\tau} & \longrightarrow \widehat{\tau} \\
\quad \widehat{\pi} & , t & \longrightarrow \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \Delta(t)) \\
\quad .1^{|\widehat{\tau}|}.\widehat{\pi} & , \widehat{\tau} & \longrightarrow \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) \\
\quad .(b_{|\widehat{\tau}|-1} \ldots b_0).\widehat{\pi} & , \widehat{\tau} \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \widehat{\tau}_i & \longrightarrow \widehat{\mathbf{focus}}_\Delta\left(\widehat{\pi}, \widehat{\tau} \bowtie_{i \in I} [o_i : \ell_i] : \widehat{\tau}_i\right) \\
 & & \quad \text{where } I = \left\{ i \;\middle|\; \begin{array}{c} 0 \leqslant i < n \\ \forall j \in \{0, \ldots, \ell_i - 1\}, b_{o_i + j} = 1 \end{array} \right\} \\
\quad .*.\widehat{\pi} & , \&_\ell(\widehat{\tau}) & \longrightarrow \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) \\
\quad .r_i.\widehat{\pi} & , \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i & \longrightarrow \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_i) \\
\quad .i.\widehat{\pi} & , \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} & \longrightarrow \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_i) \\
\} & &
\end{array}
$$

Figure 3.12: Memory-level focusing operation on types in type variable environment Δ.

We can now define syntax-based operations on memory types that access parts relevant to the represented high-level type in a generic way. The **shatter** operation, defined in Fig. 3.13, gathers all fragments and primitive types that appear within a memory type, along with their positions.

$$\mathbf{shatter}_\Delta(\widehat{\tau}) = \left\{(\widehat{\pi} \mapsto \pi \text{ as } \widehat{\tau}') \;\middle|\; \widehat{\mathbf{focus}}_\Delta\,(\widehat{\pi}, \widehat{\tau}) = (\pi \text{ as } \widehat{\tau}')\right\} \cup \left\{(\widehat{\pi} \mapsto \varepsilon \text{ as } I_\ell) \;\middle|\; \widehat{\mathbf{focus}}_\Delta\,(\widehat{\pi}, \widehat{\tau}) = I_\ell\right\}$$

Figure 3.13: Collect all fragments and atoms of a memory type.

Example 3.5 (Focus and shatter on pair lists)**.** Consider the second split branch from $\widehat{\tau}_p$ of Example 3.4, corresponding to the provenance Cons(⟨_, Nil⟩): let $\widehat{\tau} = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons.0 as } I_{32})$. We can extract all fragments from this memory type with **shatter**:

$$\mathbf{shatter}_{\Delta_{\text{list}}}(\widehat{\tau}) = \{(.[2:32] \mapsto .\text{Cons.0 as } I_{32})\}$$

Indeed, we do have $\widehat{\mathbf{focus}}_{\Delta_{\text{list}}}\,(.[2:32], \widehat{\tau}) = (.\text{Cons.0 as } I_{32})$. △

This memory focusing operation is sufficient to destruct concrete memory structures such as pointers and structs, but is undefined (for non-empty paths) on memory type constructs that refer back to a high-level type – that is, primitive (integer) encodings, fragments and splits. Fragments should be explicitely expanded as needed, rather than during focusing, so as to prevent non-termination in the presence of recursive types. As for splits, we handle them with another syntactical operation: *specialization*.

Memory type specialization, denoted $\widehat{\tau}/p$ and defined in Fig. 3.14, handles splits in a memory type $\widehat{\tau}$ by filtering out all parts of this type that do not match the pattern p, similar to high-level type specialization. However, since each split may represent arbitrarily deep provenances in different ways, it returns a *list* of branches rather than a single specialized type. Each of these branches consists of a pattern which matches precisely the high-level values it represents, and of a *split-free* memory type. In particular, $\widehat{\tau}/\_$ removes all splits from the memory type $\widehat{\tau}$ and yields the list of all its fully specialized versions.

$$t/p = \Delta(t)/p \qquad \__\ell/p = \{(p, \__\ell)\} \qquad (c)_\ell/p = \{(p, (c)_\ell)\} \qquad I_\ell/p = \{(p, I_\ell)\}$$

$$\&_\ell\,(\widehat{\tau})\,/p = \left\{(p', \&_\ell\,(\widehat{\tau}')) \;\middle|\; (p', \widehat{\tau}') \in \widehat{\tau}/p\right\}$$

$$\left(\widehat{\tau} \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{\tau}_i\right)/p = \left\{(p'', \widehat{\tau}' \ltimes r_i : \widehat{\tau}'_i) \;\middle|\; \begin{array}{c}(p', \widehat{\tau}') \in \widehat{\tau}/p \\ (p_i, \widehat{\tau}'_i) \in \widehat{\tau}_i/p \\ p'' = p' \sqcap p_0 \sqcap \ldots \sqcap p_{n-1}\end{array}\right\}$$

$$\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}\,/p = \left\{(p', \{\!\{\widehat{\tau}'_0, \ldots, \widehat{\tau}'_{n-1}\}\!\}) \;\middle|\; \begin{array}{c}(p_i, \widehat{\tau}'_i) \in \widehat{\tau}_i/p \\ p' = p_0 \sqcap \ldots \sqcap p_{n-1}\end{array}\right\} \qquad (\pi \text{ as } \widehat{\tau})/p = \{(p, (\pi \text{ as } \widehat{\tau}))\}$$

$$\text{split}\,(\widehat{\pi}_0, \ldots, \widehat{\pi}_{N-1})\left\{c_{i,0}, \ldots, c_{i,N-1} \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n\right\}/p = \bigsqcup \left\{\widehat{\tau}_i/p' \;\middle|\; \begin{array}{c}0 \leqslant i < n \\ p_i \in P_i \\ p' = p \sqcap p_i\end{array}\right\}$$

Figure 3.14: Specialization of a memory type according to a pattern in the type variable environment Δ.

Example 3.6 (Specialization of pair lists)**.** Recall the $\widehat{\tau}_p$ memory type with the type variable environment Δ_{list} from Example 3.4. Its specialization for non-empty lists, that is, for the pattern Cons(_), is:

$$\begin{array}{l}\widehat{\tau}_p\,/\,\text{Cons}(\_) = \{ \\ \quad (\text{Cons}(\langle \_, \text{Nil}\rangle) \quad , \ \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons.0 as } I_{32}),) \\ \quad (\text{Cons}(\langle \_, \text{Cons}(\_)\rangle) \ , \ \&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1.Cons.0 as } I_{32}), (.\text{Cons.1.Cons.1 as } t_p)\}\!\}) \ltimes [0:2] : (2)_2) \\ \}\end{array}$$

△

At this point, we have defined the main constituents of the Ribbitulus: high-level and memory types, high-level patterns, valuexpressions and full expressions. Recall that valuexpressions are values with the addition of path-based variable accessors. Notably, we defined two crucial operations for both high-level and memory-level objects: *specialization* by a pattern which returns one or more restricted types, and *focusing* on a path which returns a subterm of the given term. An index of all these concepts and notations is available in Fig. 3.34 at the end of this chapter.

3.1.3 Memory model

Now that the user-visible part of our language has been defined, we provide an abstraction of memory contents and a low-level representation of ribbit programs.

3.1.3.1 Memory values and expressions

After having defined memory types and their associated tools, we can define their values and expressions that represent computations on those values. Memory values, denoted $\widehat{v}$ and defined in Fig. 3.15, feature the same concrete memory structures as memory types (structs, composite words, etc., see Fig. 3.9), but differ in how pointers are represented. While a pointer memory type directly contains its pointee memory type, we use a more detailed memory model for values. A pointer memory value instead contains an *address*, denoted $a \in \mathit{Addrs}$, whose contents are accessible through a *store* ς which maps addresses to memory values. Memory *valuexpressions*, denoted $\widehat{u}$ and defined in Fig. 3.15, in the spirit of high-level valuexpressions, are similar to memory values but retain some constructs that are yet to be evaluated. There are two such constructs: pointers of the form $\&_\ell(\widehat{u})$ in which the pointee $\widehat{u}$ has not been stored and assigned an address yet, and *pivot expressions* of the form $(u : \tau \text{ as } \widehat{\tau})$. A pivot expression $(u : \tau \text{ as } \widehat{\tau})$ expresses that the high-level valuexpression u of type τ must be represented using the memory layout $\widehat{\tau}$. Within a memory valuexpression, a pivot expression acts as a placeholder for a memory value that has not been computed yet.

Memory valuexpressions

$$
\begin{aligned}
\widehat{u} \in \widehat{\mathit{ValuExprs}} ::= {} & (u : \tau \text{ as } \widehat{\tau}) && \text{(pivot expression)} \\
\mid {} & \__\ell && \text{(uninitialized } \ell\text{-bit wide word)} \\
\mid {} & (c)_\ell && (\ell\text{-bit wide constant}) \\
\mid {} & \&_\ell(\widehat{u}) && (\ell\text{-bit wide pointer to } \widehat{u}) \\
\mid {} & \&_\ell(a) && (\ell\text{-bit wide address}) \\
\mid {} & \widehat{u} \underset{0 \leqslant i < n}{\bowtie} r_i : \widehat{u}_i && \text{(composite word)} \\
\mid {} & \{\!\{\widehat{u}_0, \ldots, \widehat{u}_{n-1}\}\!\} && (n\text{-field struct})
\end{aligned}
$$

Memory values

$$
\widehat{v} \in \widehat{\mathit{Values}} ::= \__\ell \mid (c)_\ell \mid \&_\ell(a) \mid \widehat{v} \underset{0 \leqslant i < n}{\bowtie} r_i : \widehat{v}_i \mid \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\}
$$

$$
\varsigma : \mathit{Addrs} \to \widehat{\mathit{Values}} \qquad \text{(memory store)}
$$

Figure 3.15: Memory values and valuexpressions.

Similar to the memory focusing operation defined on memory types, we define a memory focusing operation on memory valuexpressions in Fig. 3.16. It differs from memory focusing on types in that it depends on a store ς to allow for focusing below addresses.

$$\widehat{\mathbf{focus}}_\varsigma(\varepsilon, \widehat{u}) = \widehat{u} \qquad \widehat{\mathbf{focus}}_\varsigma\left(.r_i.\widehat{\pi}, \widehat{u} \underset{0\leqslant i<n}{\bowtie} r_i : \widehat{u}_i\right) = \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{u}_i) \qquad \widehat{\mathbf{focus}}_\varsigma\left(.1^{|\widehat{u}|}.\widehat{\pi}, \widehat{u}\right) = \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{u})$$

$$\frac{m = b_{|\widehat{u}|-1}.....b_0 \qquad I = \left\{ i \;\middle|\; \begin{array}{c} 0 \leqslant i < n \\ \forall j \in \{0,\dots,\ell_i - 1\}, b_{o_i+j} = 1 \end{array} \right\}}{\widehat{\mathbf{focus}}_\varsigma\left(.m.\widehat{\pi}, \widehat{u} \underset{0\leqslant i<n}{\bowtie} [o_i : \ell_i] : \widehat{u}_i\right) = \widehat{\mathbf{focus}}_\varsigma\left(\widehat{\pi}, \widehat{u} \underset{i\in I}{\bowtie} [o_i : \ell_i] : \widehat{u}_i\right)}$$

$$\widehat{\mathbf{focus}}_\varsigma(.*.\widehat{\pi}, \&_\ell(\widehat{u})) = \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{u}) \qquad \frac{(a \mapsto \widehat{v}) \in \varsigma}{\widehat{\mathbf{focus}}_\varsigma(.*.\widehat{\pi}, \&_\ell(a)) = \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{v})}$$

$$\widehat{\mathbf{focus}}_\varsigma(.i.\widehat{\pi}, \{\!\{\widehat{u}_0, \dots, \widehat{u}_{n-1}\}\!\}) = \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{u}_i)$$

Figure 3.16: Memory-level focusing operation on valuexpressions in the store ς.

Example 3.7 (Pair list memory valuexpression)**.** Recall the memory type $\widehat{\tau}_p$ defined for the τ_{list} ADT from Example 3.4. The following memory valuexpression is a pivot expression that requests the representation of the high-level value Cons(42, Nil) according to the layout $\widehat{\tau}_p$:

$$(\text{Cons}(42, \text{Nil}) : \tau_{list} \text{ as } \widehat{\tau}_p)$$

After evaluation (which will be defined in Section 3.3.2), we would obtain the following memory value (and an empty store):

$$\widehat{v} = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

This memory value follows the structure of the specified memory type $\widehat{\tau}_p$, with each fragment instanciated with the supplied concrete value. Focusing on it with the memory path .[2 : 32] yields the encoded integer value of the first element:

$$\widehat{\mathbf{focus}}_\varnothing(.[2:32], \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}) = (42)_{32}$$

△

We can now define full memory expressions, which capture all stages from a source program down to its fully evaluated form, that is, a memory value. To this end, full memory expressions, denoted $\widehat{e}$ and defined in Fig. 3.17, include all high-level full expressions, all memory valuexpressions, as well as hybrid let-expressions that fit neither existing grammar but may arise during evaluation.

$$\begin{array}{lll} \widehat{e} \in \widehat{Exprs} ::= e \in Exprs & & \text{(high-level expression)} \\ \quad | \; \widehat{u} \in \widehat{ValuExprs} & & \text{(memory valuexpression)} \\ \quad | \; \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = \widehat{e} \ \mathsf{in}\ e & & \text{(intermediate let-bind form)} \end{array}$$

Figure 3.17: Full memory expressions.

Example 3.8 (Full memory expression for pair lists)**.** The following memory expression binds x to the memory value $\widehat{v}$ from Example 3.7 and accesses its integer value encoded on 32 bits using the pivot expression (x.Cons.0 : I_{32} as I_{32}).

$$\widehat{e} = \mathsf{let}\ x : \tau_{list} \text{ as } \widehat{\tau}_p = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32} \ \mathsf{in}\ (x.\text{Cons}.0 : I_{32} \text{ as } I_{32})$$

△

3.1.3.2 Memory patterns: shapes of memory contents

$$
\begin{aligned}
\widehat{p} \in \widehat{\mathit{Patterns}} ::= {} & \__\ell && (\ell\text{-bit wide wildcard}) \\
\mid {} & (c)_\ell && (\ell\text{-bit wide constant}) \\
\mid {} & \&_\ell\,(\widehat{p}) && (\ell\text{-bit wide pointer}) \\
\mid {} & \widehat{p} \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{p}_i && (\text{composite word}) \\
\mid {} & \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\} && (\text{struct})
\end{aligned}
$$

Figure 3.18: Memory patterns. Sets of memory patterns are denoted $\widehat{P}$.

Similarly to high-level patterns, we define a notion of *memory patterns*, denoted $\widehat{p}$ and defined in Fig. 3.18, which describe the shape of a given memory type of valuexpression. Empty word memory patterns act as (sized) *wildcards*, in that $\__\ell$ matches any memory contents of size ℓ. The function **shape_of**, defined in Fig. 3.19, returns a memory pattern corresponding to the shape of a memory type or valuexpression; it mostly follows the syntax of memory constructs. For constructs that refer to parts of a high-level type or value – that is, primitive types, fragments, splits and pivot expressions – we use an appropriately sized wildcard. Indeed, we consider such constructs as "black boxes" whose size is known but whose precise contents are not determined yet.

$\mathbf{shape\_of}_\Delta\{$

t	$\longrightarrow$	$\mathbf{shape\_of}_\Delta(\Delta(t))$
$\__\ell$	$\longrightarrow$	$\__\ell$
$(c)_\ell$	$\longrightarrow$	$(c)_\ell$
$\&_\ell\,(\widehat{\tau})$	$\longrightarrow$	$\&_\ell\,(\mathbf{shape\_of}_\Delta(\widehat{\tau}))$
$\widehat{\tau} \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{\tau}_i$	$\longrightarrow$	$\mathbf{shape\_of}_\Delta(\widehat{\tau}) \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{p}_i$
$\{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$	$\longrightarrow$	$\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$ where $\widehat{p}_i = \mathbf{shape\_of}_\Delta(\widehat{\tau}_i)$
$\widehat{\tau}$	$\longrightarrow$	$\__{\lvert\widehat{\tau}\rvert}$

$\}$

(a) Shape of a memory type in the type variable environment Δ

$\mathbf{shape\_of}_\varsigma\{$

$\__\ell$	$\longrightarrow$	$\__\ell$
$(c)_\ell$	$\longrightarrow$	$(c)_\ell$
$\&_\ell\,(\widehat{u})$	$\longrightarrow$	$\&_\ell\left(\mathbf{shape\_of}_\varsigma(\widehat{u})\right)$
$\&_\ell\,(a)$	$\longrightarrow$	$\&_\ell\left(\mathbf{shape\_of}_\varsigma(\varsigma(a))\right)$
$\widehat{u} \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{u}_i$	$\longrightarrow$	$\mathbf{shape\_of}_\varsigma(\widehat{u}) \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{p}_i$
$\{\!\{\widehat{u}_0, \ldots, \widehat{u}_{n-1}\}\!\}$	$\longrightarrow$	$\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$ where $\widehat{p}_i = \mathbf{shape\_of}_\varsigma(\widehat{u}_i)$
$(u : \tau \text{ as } \widehat{\tau})$	$\longrightarrow$	$\__{\lvert\widehat{\tau}\rvert}$

$\}$

(b) Shape of a memory valuexpression in the store ς

Figure 3.19: Memory pattern capturing the shape of a memory type or valuexpression.

Example 3.9 (Shapes of pair lists)**.** The shape of the memory type $\widehat{\tau}_p$ from Example 3.4 is $\mathbf{shape\_of}(\widehat{\tau}_p) = \__{64}$. It fits all possible values: due to the toplevel split in $\widehat{\tau}_p$, the only available information is the maximal size of its values – here, all of its values are necessarily 64 bits wide.

On the other hand, the shape of the memory value $\widehat{v}$ from Example 3.7 is more precise since all elements that were unevaluated in $\widehat{\tau}_p$ are now fully determined:

$$\mathbf{shape\_of}(\widehat{v}) = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

△

3.2 Typing and validity

To ensure the correctness of a given memory specification, we consider two complementary notions: intrinsic validity and well-kindedness of a memory type, and agreement between high-level and memory

types.

3.2.1 Kinding and validity of memory types

The intrisic validity of memory types relies on two judgments. The notion of *well-kindedness*, denoted $\Delta \vDash \widehat{\tau}$, refers to memory types passing both of these judgments. While it would be possible to define a single judgment, determining a kind and checking validity at the same time, this approach quickly leads to cycles in derivation trees, which we would rather avoid. Using two separate judgments lets us break the recursive cycle and still cover the entire structure of a (possibly recursive) memory type.

$$\textsc{VTyVar}\ \frac{t \in \mathrm{dom}(\Delta) \qquad \Delta(t) \in \widehat{\mathit{Types}}}{\Delta \vDash t\ \mathrm{valid}} \qquad \textsc{VConstant}\ \frac{0 \leqslant c < 2^{\ell}}{\Delta \vDash (c)_{\ell}\ \mathrm{valid}} \qquad \textsc{VWord}\ \Delta \vDash \__{\ell}\ \mathrm{valid} \qquad \textsc{VPrimitive}\ \Delta \vDash I_{\ell}\ \mathrm{valid} \qquad \textsc{VPointer}\ \frac{\Delta \vDash \widehat{\tau}\ \mathrm{valid}}{\Delta \vDash \&_{\ell}(\widehat{\tau})\ \mathrm{valid}}$$

$$\textsc{VComposite}\ \frac{\Delta \vDash \widehat{\tau}\ \mathrm{valid} \qquad \Delta \vDash \widehat{\tau}_i\ \mathrm{valid} \qquad i < j \Rightarrow o_i + \ell_i \leqslant o_j}{\Delta \vDash \widehat{\tau} \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \widehat{\tau}_i\ \mathrm{valid}} \qquad \textsc{VStruct}\ \frac{\Delta \vDash \widehat{\tau}_i\ \mathrm{valid}}{\Delta \vDash \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}\ \mathrm{valid}} \qquad \textsc{VFragment}\ \frac{\Delta \vDash \widehat{\tau}\ \mathrm{valid}}{\Delta \vDash (\pi\ \mathrm{as}\ \widehat{\tau})\ \mathrm{valid}}$$

$$\textsc{VSplit}\ \frac{\begin{array}{c}\Delta \vDash \widehat{\tau}_i\ \mathrm{valid} \\ \exists \ell, \forall (p, \widehat{\tau}) \in \widehat{\tau}_i \,/\, \_, \widehat{\mathbf{focus}}\,(\widehat{\pi}_j, \widehat{\tau}) = (c_{i,j})_{\ell} \qquad \forall p \in P_i, \forall p' \in P_j, i \neq j \Rightarrow p \sqcap p' \text{ is undefined}\end{array}}{\Delta \vDash \mathrm{split}\,(\widehat{\pi}_1, \ldots, \widehat{\pi}_N)\left\{ c_{i,1}, \ldots, c_{i,N}\ \mathrm{from}\ P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 1 \leqslant i \leqslant n \right\}\ \mathrm{valid}}$$

Figure 3.20: Validity judgment on memory types.

The *validity* judgment, denoted $\Delta \vDash \widehat{\tau}\,\mathrm{valid}$ and defined in Fig. 3.20, explores a memory type by iterating over its entire inductive structure, stopping at type variables. It ensures that every specified construct "makes sense" individually. Its main purpose is to check the good formation of memory type constructs: the VTyVar rule checks that each type variable is bound to a memory type, while the VComposite rule ensures that composite word bit ranges do not overlap. For splits (VSplit rule), we check that each branch indeed contains the specified discriminant values at their respective positions and that branch provenances do not overlap each other. In order to fully check the validity of a recursive type, we simply iterate over each memory type bound in the type variable environment and check their validity.

Example 3.10 (Invalid memory type: C unions)**.** This validity judgment already rejects non-trivial user mistakes. Let us emulate a traditional C union layout for the τ_{zarith} type defined in Section 2.2 which takes up as much space as the largest variant (Large):

$$\widehat{\tau}_{\mathrm{bad}} = \__{128} \ltimes [0:63] : (.\mathrm{Small}\ \mathrm{as}\ I_{63}) \ltimes [0:128] : (.\mathrm{Large}\ \mathrm{as}\ I_{128})$$

$\widehat{\tau}_{\mathrm{bad}}$ is not valid: the VComposite rule does not apply because $[0:63]$ and $[0:128]$ overlap. Since it lacks distinguishing data to use as a discriminant, this layout is not expressable as a split. △

Example 3.11 (Valid memory type for lists)**.** Recall the following memory type from Example 3.4, in the type variable environment Δ_{list}:

$$\widehat{\tau}_c = \mathrm{split}\,(.[0:1])\left\{ \begin{array}{lll} 1 & \mathrm{from}\ \mathrm{Nil} & \Rightarrow \__{64} \ltimes [0:1] : (1)_1 \\ 0 & \mathrm{from}\ \mathrm{Cons}(\_) & \Rightarrow \&_{64}(\widehat{\tau}_s) \ltimes [0:1] : (0)_1 \end{array} \right\}$$

where

$$\widehat{\tau}_s = \{\!\{(.\mathrm{Cons}.0\ \mathrm{as}\ I_{32}), (.\mathrm{Cons}.1\ \mathrm{as}\ t_c)\}\!\}$$

We show that $\widehat{\tau}_c$ is a valid memory type:

$$\textsc{VStruct}\ \frac{\textsc{VFragment}\ \dfrac{\textsc{VPrimitive}\ \Delta \vDash I_{32}\ \text{valid}}{\Delta \vDash (.\text{Cons.0 as } I_{32})\ \text{valid}} \qquad \textsc{VFragment}\ \dfrac{\textsc{VTyVar}\ \dfrac{(t_c \mapsto \widehat{\tau}_c) \in \Delta \qquad \widehat{\tau}_c \in \widehat{\mathit{Types}}}{\Delta \vDash t_c\ \text{valid}}}{\Delta \vDash (.\text{Cons.1 as } t_c)\ \text{valid}}}{\Delta \vDash \widehat{\tau}_s\ \text{valid}}$$

$$\textsc{VSplit}\ \frac{\textsc{VComposite}\ \dfrac{\textsc{VWord}\ \dfrac{}{\Delta \vDash \__{64}\ \text{valid}} \quad \textsc{VConstant}\ \dfrac{0 \leqslant 1 < 2}{\Delta \vDash (1)_1\ \text{valid}}}{\Delta \vDash \__{64} \ltimes [0:1] : (1)_1\ \text{valid}} \qquad \textsc{VComposite}\ \dfrac{\textsc{VPointer}\ \dfrac{\Delta \vDash \widehat{\tau}_s\ \text{valid}}{\Delta \vDash \&_\ell(\widehat{\tau}_s)\ \text{valid}} \quad \textsc{VConstant}\ \dfrac{0 \leqslant 0 < 2}{\Delta \vDash (0)_1\ \text{valid}}}{\Delta \vDash \&_{64}(\widehat{\tau}_s) \ltimes [0:1] : (0)_1\ \text{valid}}}{\widehat{\textbf{focus}}\,(.[0:1], \__{64} \ltimes [0:1] : (1)_1) = (1)_1 \qquad \widehat{\textbf{focus}}\,(.[0:1], \&_{64}(\widehat{\tau}_s) \ltimes [0:1] : (0)_1) = (0)_1 \qquad \text{Nil} \sqcap \text{Cons}(\_)\ \text{is undefined}}{\Delta \vDash \widehat{\tau}_c\ \text{valid}}$$

△

The *kinding* judgment, denoted $\Delta \vDash \widehat{\tau} : \widehat{\kappa}$ and defined in Fig. 3.21, checks that parts of a memory type "make sense" in relation with each other. For instance, KComposite ensures that composite word bit ranges indeed fit within unused bits of the base memory type and do not overlap each other. To this end, it assigns a *kind*, denoted $\widehat{\kappa}$, to each memory type. A kind is either **Block**, which represents structs and can never appear inside a composite word, or **Word**(m) representing words of the same width as m, where m is a bit mask in which zeroes indicate bits that are necessarily free (not used for storing data by the memory type). The kinding judgment, unlike the validity judgment, follows type variables but does not recursively explore pointers. Indeed, only the address alignment (as opposed to the full kind) of a memory type is needed to determine the kind of a pointer to this type. In the actual judgment, we only informally state this address alignment criterion: since the actual set of available bits in pointers is highly architecture/OS-dependent, we leave this information out of the validity and kinding judgments. For now, this means that the validity of such optimizations is left up to the user; a more satisfactory solution would be to develop architecture/OS-specific extensions for ribbit to check it automatically.

$$\widehat{\kappa} ::= \textbf{Word}(m) \mid \textbf{Block}$$

$$\textsc{KTyVar}\ \frac{\Delta \vDash \Delta(t) : \widehat{\kappa}}{\Delta \vDash t : \widehat{\kappa}} \qquad \textsc{KConstant}\ \Delta \vDash (c)_\ell : \textbf{Word}(1^\ell) \qquad \textsc{KWord}\ \Delta \vDash \__\ell : \textbf{Word}(0^\ell) \qquad \textsc{KPrimitive}\ \Delta \vDash I_\ell : \textbf{Word}(1^\ell)$$

$$\textsc{KPointer}\ \frac{m \text{ sets the address bits of any } \ell\text{-bit wide } \widehat{\tau} \text{ pointer}}{\Delta \vDash \&_\ell(\widehat{\tau}) : \textbf{Word}(m)} \qquad \textsc{SubKinding}\ \frac{\Delta \vDash \widehat{\tau} : \textbf{Word}(m.0.m')}{\Delta \vDash \widehat{\tau} : \textbf{Word}(m.1.m')}$$

$$\textsc{KComposite}\ \frac{\Delta \vDash \widehat{\tau} : \textbf{Word}(m) \qquad \Delta \vDash \widehat{\tau}_i : \textbf{Word}(m_i) \qquad m'_i = \overbrace{0\ldots0}^{|\widehat{\tau}|\ \text{bits}}\ \underbrace{m_i}_{\ell_i\ \text{bits}}\ \underbrace{0\ldots0}_{o_i\ \text{bits}}}{\Delta \vDash \widehat{\tau} \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \widehat{\tau}_i : \textbf{Word}(m \vee m'_0 \vee \cdots \vee m'_{n-1})} \qquad \textsc{KStruct}\ \Delta \vDash \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} : \textbf{Block}$$

$$\textsc{KFragment}\ \frac{\Delta \vDash \widehat{\tau} : \widehat{\kappa}}{\Delta \vDash (\pi \text{ as } \widehat{\tau}) : \widehat{\kappa}} \qquad \textsc{KSplit}\ \frac{\Delta \vDash \widehat{\tau}_i : \widehat{\kappa}}{\Delta \vDash \text{split}\,(\widehat{\pi}_1, \ldots, \widehat{\pi}_N)\,\{\, c_{i,1}, \ldots, c_{i,N} \ \text{ from } P_i \Rightarrow \widehat{\tau}_i \mid 1 \leqslant i \leqslant n\,\} : \widehat{\kappa}}$$

Figure 3.21: Memory kinds $\widehat{\kappa}$ and kinding judgment $\Delta \vDash \widehat{\tau} : \widehat{\kappa}$, used on *valid* memory types.

Most of the rules are fairly immediate by direct induction. The KComposite proceeds by checking non-overlapping of masks. The SubKinding rule lets us relax a **Word** kind that has been assigned to a memory type, by "forgetting" that a given bit is unused. This is useful to unify the kinds of all branches in the KSplit rule.

Example 3.12 (Well-kinded memory type for lists)**.** Recall the following memory type from Example 3.4, in the type variable environment Δ_{list}:

$$\widehat{\tau}_c = \text{split}\,(.[0:1])\left\{\begin{array}{lll} 1 \text{ from } \text{Nil} & \Rightarrow & \_{}_{64} \ltimes [0:1] : (1)_1 \\ 0 \text{ from } \text{Cons}(\_) & \Rightarrow & \&_{64}\,(\widehat{\tau}_s) \ltimes [0:1] : (0)_1 \end{array}\right\}$$

where

$$\widehat{\tau}_s = \{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1 as } t_c)\}\!\}$$

We have $|\widehat{\tau}_s| = |I_{32}| + |\widehat{\tau}_c| = 32 + 64 = 96$. Assuming that the address alignment of a struct is equal to its size *in bytes*, we can assert that the two lowest bits of any pointer to a 96-bits-wide struct are free to use. Using the KPointer rule, we can therefore assign the following kind to $\&_{64}\,(\widehat{\tau}_s)$:

$$\Delta \vDash \&_{64}\,(\widehat{\tau}_s) : \textbf{Word}(1^{62}00)$$

We can now show that $\widehat{\tau}_c$ is of kind $\textbf{Word}(1^{62}01)$. Note how the last KComposite rule on the right marks the last bit in the kind as used, manifesting the bit-stealing.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vDash \_{}_{64} : \textbf{Word}(0^{64})}\text{KWord} \quad \dfrac{}{\Delta \vDash (1)_1 : \textbf{Word}(1)}\text{KConstant}}{\Delta \vDash \_{}_{64} \ltimes [0:1] : (1)_1 : \textbf{Word}(0^{63}1)}\text{KComposite}}{\Delta \vDash \_{}_{64} \ltimes [0:1] : (1)_1 : \textbf{Word}(1^{62}01)}\text{SubKinding} \quad \dfrac{\Delta \vDash \&_{64}\,(\widehat{\tau}_s) : \textbf{Word}(1^{62}00) \quad \dfrac{}{\Delta \vDash (0)_1 : \textbf{Word}(1)}\text{KConstant}}{\Delta \vDash \&_{64}\,(\widehat{\tau}_s) \ltimes [0:1] : (0)_1 : \textbf{Word}(1^{62}01)}\text{KComposite}}{\Delta \vDash \widehat{\tau}_c : \textbf{Word}(1^{62}01)}\text{KSplit}$$

Δ

We can now define *well-formed* type variable environments, in which every bound type is valid and well-kinded.

Definition 3.1 (well-formed type variable environments)**.** A type variable environment Δ is *well-formed*, and we write $\vDash \Delta$, if and only if every memory type is valid and well-kinded and all type variables that appear in high-level (resp. memory) types are bound to a high-level (resp. memory) type, i.e.:

- $\forall (t \mapsto \tau) \in \Delta, \forall t' \in \mathit{TyVars}, \forall \pi \in \mathit{Paths}$ such that $\textbf{focus}\,(\pi, \tau) = t', \exists \tau' \in \mathit{Types}, (t' \mapsto \tau') \in \Delta$
- $\forall (t \mapsto \widehat{\tau}) \in \Delta, \Delta \vDash \widehat{\tau}$ valid $\wedge\ \exists \widehat{\kappa}, \Delta \vDash \widehat{\tau} : \widehat{\kappa}$

3.2.2 Agreement between ADTs and memory layouts

Now that we have defined validity criteria for memory types on their own, we can state the relationship between ADTs and their memory layout specifications – that is, whether a given memory type properly represents, or *agrees with*, a given high-level type. This agreement relation is based on four criteria, which we formally state in Definition 3.2. *Coverage* ensures that every piece of data from the high-level type appears within the memory type, as an arbitrary combination of fragments and primitive encodings. Note that it is acceptable to split a subterm into any number of pieces and scatter them arbitrarily across the memory type, as long as every piece appears somewhere within the memory type. *Distinguishability* ensures that the precise provenance of any given high-level value is always identifiable from its memory representation, by inspecting a combination of split discriminant locations. Again, any configuration of split locations, values and branch provenances is acceptable, as long as every provenance is distinguishable from incompatible other provenances. The two remaining criteria (fragment and branch coherence) simply propagate coverage and distinguishability through the entire inductive structures of both high-level and memory types.

Definition 3.2 (Agreement)**.** Let τ a high-level type and $\widehat{\tau}$ a memory type considered in a type variable environment Δ. We say that $\widehat{\tau}$ *represents* τ, or *agrees* with τ, and we write $\mathbf{agree}_\Delta(\tau, \widehat{\tau})$, if either τ and $\widehat{\tau}$ are identical primitive types (i.e., $\tau = \widehat{\tau} = \mathrm{I}_\ell$) or all of the following conditions hold:

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

Split branches are valid representations of high-level subtypes. (Branch Coherence)
For all $\widehat{\pi}$ such that $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) = \mathrm{split}(\ldots)\left\{\ldots \text{ from } \mathrm{P}_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n\right\}$, for each branch $i \in \{0, \ldots, n-1\}$ and each $p \in \mathrm{P}_i$ such that $\Delta \vdash p : \tau$, $\widehat{\tau}_i$ agrees with τ/p. Furthermore, for every pattern p of type τ, there exists a branch $i \in \{0, \ldots, n-1\}$ and $p' \in \mathrm{P}_i$ such that $p \sqcap p'$ is defined.

All data from the high-level type is represented within the memory type. (Coverage)
For every high-level path π that leads to a single bit in τ (i.e., $\mathbf{focus}_\Delta(\pi, \tau) = \mathrm{I}_1$), $\widehat{\tau}$ *covers* π: every memory type $\widehat{\tau}' \in \widehat{\tau}/\pi$ contains a fragment (or primitive type) for a position π_0 prefix of π. More precisely, there exist high-level and memory paths π_0 and $\widehat{\pi}_0$ such that $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}_0, \widehat{\tau}') = (\pi_0 \text{ as } \widehat{\tau}'')$ and either $\pi = \pi_0.\pi'$ or $\pi = \pi'.[o : \ell]$, $\pi_0 = \pi'.[o_0 : \ell_0]$ and $o_0 \leqslant o \leqslant o + \ell \leqslant o_0 + \ell_0$ (or $\pi = \varepsilon$, $\tau = \mathrm{I}_\ell$ and $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}_0, \widehat{\tau}') = \mathrm{I}_\ell$).

Memory types provide a way to tell incompatible patterns apart. (Distinguishability)
For every high-level path π that leads to a sum in τ (i.e., $\mathbf{focus}_\Delta(\pi, \tau) = \mathrm{K}_0(\tau_0) \mid \cdots \mid \mathrm{K}_{n-1}(\tau_{n-1})$), for every pair of distinct constructors in this sum $(\mathrm{K}_i, \mathrm{K}_j)$ (with $0 \leqslant i \neq j < n$), $\widehat{\tau}$ *distinguishes* between K_i and K_j. More precisely, $\widehat{\tau}/\pi.\mathrm{K}_i \neq \varnothing$, $\widehat{\tau}/\pi.\mathrm{K}_j \neq \varnothing$ and for any $(p_i, \widehat{\tau}_i) \in \widehat{\tau}/\pi.\mathrm{K}_i$ and $(p_j, \widehat{\tau}_j) \in \widehat{\tau}/\pi.\mathrm{K}_j$, there exists a memory path $\widehat{\pi}$ such that either $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_i) = \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_j) = (\pi_0 \text{ as } \widehat{\tau}')$ with π_0 a prefix of π, or $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_i) = (c_i)_\ell$, $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_j) = (c_j)_\ell$ and $c_i \neq c_j$.

As a first example, we show that our memory layout specification for lists agrees with their ADT.

Example 3.13 (Agreement for lists)**.** Recall Δ_{list}, τ_{list} and $\widehat{\tau}_c$ from our running example:

$$\Delta_{\mathrm{list}} = \{t_{\mathrm{list}} \mapsto \tau_{\mathrm{list}}, t_c \mapsto \widehat{\tau}_c, t_p \mapsto \widehat{\tau}_p\} \qquad \tau_{\mathrm{list}} = \mathrm{Nil} \mid \mathrm{Cons}(\langle \mathrm{I}_{32}, t_{\mathrm{list}} \rangle)$$

$$\widehat{\tau}_c = \mathrm{split}(.[0:1])\left\{\begin{array}{lll} 1 \text{ from } \mathrm{Nil} & \Rightarrow & \_{}_{64} \ltimes [0:1] : (1)_1 \\ 0 \text{ from } \mathrm{Cons}(\_) & \Rightarrow & \&_{64}(\{\!\{(.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), (.\mathrm{Cons}.1 \text{ as } t_c)\}\!\}) \ltimes [0:1] : (0)_1 \end{array}\right\}$$

To establish agreement between τ_{list} and $\widehat{\tau}_c$, we first show that both branches agree with their specialized ADT counterparts. As τ_{list} is a recursive type, we admit that the type variables forming the recursive node agree with each other, i.e., $\mathbf{agree}_{\Delta_{\mathrm{list}}}(t_{\mathrm{list}}, t_c)$.

- Nil branch: we show that $\mathbf{agree}_{\Delta_{\mathrm{list}}}(\mathrm{Nil}, \_{}_{64} \ltimes [0:1] : (1)_1)$. All criteria are immediate, since there are no fragments or splits in this memory type and the high-level type Nil contains no primitive data and only has one constructor.
- Cons(_) branch: let $\widehat{\tau}_{\mathrm{Cons}} = \&_{64}(\{\!\{(.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), (.\mathrm{Cons}.1 \text{ as } t_c)\}\!\}) \ltimes [0:1] : (0)_1$. We show that $\mathbf{agree}_{\Delta_{\mathrm{list}}}(\mathrm{Cons}(\langle \mathrm{I}_{32}, t_{\mathrm{list}} \rangle), \widehat{\tau}_{\mathrm{Cons}})$.

  Fragment coherence: $\mathbf{agree}(\mathrm{I}_{32}, \mathrm{I}_{32})$ is immediate from the base case of Definition 3.2. The second fragment corresponds to the recursive node: $\mathbf{agree}_{\Delta_{\mathrm{list}}}(t_{\mathrm{list}}, t_c)$.

  Branch coherence: immediate since there are no splits in this memory type.

  Coverage: the primitive data in this high-level type consist of the I_{32} subterm at position .Cons.0, which is covered by the fragment $(.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})$, and of all primitive contents of the t_{list} subterm at position .Cons.1, which is covered by the fragment $(.\mathrm{Cons}.1 \text{ as } t_c)$.

  Distinguishability: immediate since there are no sums with more than one constructor in this type, except below the recursive node.

We can now show that we have $\mathbf{agree}_{\Delta_{\mathrm{list}}}(\tau_{\mathrm{list}}, \widehat{\tau}_c)$.

Fragment coherence: immediate since there are no fragments or primitive types outside of the toplevel split (i.e., $\mathbf{shatter}_{\Delta_{\text{list}}}(\widehat{\tau}_c) = \varnothing$).

Branch coherence: as shown above, we have $\mathbf{agree}_{\Delta_{\text{list}}}(\text{Nil}, \_{64} \ltimes [0:1] : (1)_1)$ and $\mathbf{agree}_{\Delta_{\text{list}}}(\text{Cons}(\langle I_{32}, t_{\text{list}}\rangle), \widehat{\tau}_{\text{Cons}})$.

Coverage: the coverage criterion for the full type follows from coverage for the Cons(_) branch.

Distinguishability: $\widehat{\tau}_c$ distinguishes between Nil and Cons, thanks to its split discriminant at position $.[0:1]$. Indeed, we have $\widehat{\mathbf{focus}}\,(.[0:1], \_{64} \ltimes [0:1] : (1)_1) = (1)_1$, $\widehat{\mathbf{focus}}\,(.[0:1], \widehat{\tau}_{\text{Cons}}) = (0)_1$ and $1 \neq 0$.

△

The rest of this section is dedicated to counter-examples, with memory types that disagree with τ_{list}.

Example 3.14 (Unit representation)**.** Consider a primitive high-level type I_ℓ and the constant memory type $(42)_\ell$. This memory type does not meet the coverage criterion for I_ℓ, because no fragment nor primitive type appears in it. Indeed, this type represents all high-level values as the constant 42 on ℓ bits. Therefore, the memory type

$$\text{split}\,(.[0:1])\left\{\begin{array}{lll} 1 \text{ from Nil} & \Rightarrow & \_{64} \ltimes [0:1] : (1)_1 \\ 0 \text{ from Cons}(\_) & \Rightarrow & \&_{64}\,(\{\!\{(.\text{Cons.0 as } (42)_\ell), (.\text{Cons.1 as t})\}\!\}) \ltimes [0:1] : (0)_1 \end{array}\right\}$$

does not meet the fragment coherence criterion for τ_{list}, because it does not encode its primitive subtype $\mathbf{focus}\,(.\text{Cons.0}, \tau_{\text{list}}) = I_{32}$. △

Example 3.15 (Non-coverage: tags without payload)**.** A simple tag without payloads, similar to C enums, is not sufficient to encode arbitrary sum types with non-unit variants, as it does not meet the coverage criterion: $\text{split}\,(\varepsilon)\left\{\begin{array}{lll} 0 \text{ from Nil} & \Rightarrow & (0)_{32} \\ 1 \text{ from Cons}(\_) & \Rightarrow & (1)_{32} \end{array}\right\}$ does not cover τ_{list} since it does not represent its subterms at positions .Cons.0 and .Cons.1. △

Example 3.16 (Non-distinguishability)**.** Let $\widehat{\tau} = \{\!\{(.\text{Cons.0 as } I_{32}), \&_{32}\,((.\text{Cons.1 as t}))\}\!\}$ with the type variable t mapped to $\widehat{\tau}$. It includes the Cons constructor's subterms in distinct struct fields but provides no way to distinguish Nil from Cons values. $\widehat{\tau}$ does not meet the distinguishability criterion for τ_{list}: we have $\widehat{\tau}/\text{Nil} = \{(\text{Nil}, \widehat{\tau})\}$ and $\widehat{\tau}/\text{Cons}(\_) = \{(\text{Cons}(\_), \widehat{\tau})\}$; it lacks a discriminant that differs between Nil and Cons(_). △

3.2.3 Typing for high-level objects

3.2.3.1 High-level typing judgment

We now define a typing judgment for high-level valuexpressions and patterns, which are accordingly typed by *high-level* types (ADTs). Even though this judgment only deals with high-level values and types, its environments include both high-level and memory-level bindings, so as to ease the definition of a typing judgment for full expressions which we cover in Section 3.2.3.2. In the following definitions, the meta-syntactical variable θ denotes a high-level object which is either a pattern or a valuexpression, i.e., $\theta \in \mathit{ValuExprs} \cup \mathit{Patterns}$. The type variable environment $\Delta : \mathit{TyVars} \to \mathit{Types} \cup \widehat{\mathit{Types}}$ maps each defined type variable to either a high-level *or* memory type, although we will only use bindings to high-level types here. Similarly, the typing environment $\Gamma : \mathit{Vars} \to \mathit{Types} \times \widehat{\mathit{Types}}$ maps each defined variable to a pair of a high-level *and* a memory type, although we will only consider the former here. The actual high-level typing judgment is denoted $\Delta, \Gamma \vdash \theta : \tau$ and defined in Fig. 3.22. For patterns, which by definition never contain variables, we may omit the typing environment and write $\Delta \vdash p : \tau$.

$$\textsc{HLTTypeVar}\ \frac{(t \mapsto \tau) \in \Delta \qquad \Delta, \Gamma \vdash \theta : \tau}{\Delta, \Gamma \vdash \theta : t} \qquad \textsc{HLTVariable}\ \frac{(x : \tau \text{ as } \widehat{\tau}) \in \Gamma}{\Delta, \Gamma \vdash x.\pi : \textbf{focus}\ (\pi, \tau)} \qquad \textsc{HLTWildcard}\ \Delta, \Gamma \vdash \_ : \tau \qquad \textsc{HLTConstant}\ \frac{0 \leqslant c < 2^{\ell}}{\Delta, \Gamma \vdash c : I_{\ell}}$$

$$\textsc{HLTTuple}\ \frac{\Delta, \Gamma \vdash \theta_i : \tau_i}{\Delta, \Gamma \vdash \langle \theta_0, \ldots, \theta_{n-1} \rangle : \langle \tau_0, \ldots, \tau_{n-1} \rangle} \qquad \textsc{HLTConstr}\ \frac{\exists i \in \{0, \ldots, n-1\}, K = K_i \qquad \Delta, \Gamma \vdash \theta : \tau_i}{\Delta, \Gamma \vdash K(\theta) : K_0(\tau_0) \mid \ldots \mid K_{n-1}(\tau_{n-1})}$$

Figure 3.22: Typing judgment for patterns and valuexpressions ($\theta \in \mathit{ValuExprs} \cup \mathit{Patterns}$).

Example 3.17 (High-level typing for lists)**.** Recall the high-level type τ_{list} in the type variable environment Δ_{list} from Example 3.1: $\tau_{\text{list}} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{\text{list}} \rangle)$. Consider the high-level value $\text{Cons}(\langle 42, \text{Nil} \rangle)$. We show that this value is of type τ_{list}, using the HLConstr, HLTuple, HLTTypeVar and HLConstant typing rules.

Now consider the valuexpression x.Cons.0 in the following typing environment: $\Gamma = \{x : \tau_{\text{list}} \text{ as } \widehat{\tau}_c\}$, where $\widehat{\tau}_c$ is one of the two memory types defined in Example 3.4 (here, it does not matter which layout we pick as long as it agrees with τ_{list}). We type the valuexpression x.Cons.0 by applying the HLTVariable rule; since $\textbf{focus}\ (.\text{Cons}.0, \tau_{\text{list}}) = t_{\text{list}}$, we have $\Delta_{\text{list}}, \Gamma \vdash x.\text{Cons}.0 : t_{\text{list}}$. △

Definition 3.3 (well-typed value environments)**.** Let Δ a well-formed type variable environment and Γ a well-formed typing environment in Δ. A value binding environment σ is *well-typed* in Δ and Γ, and we write $\Delta, \Gamma \vdash \sigma$, if and only if $\text{dom}(\sigma) = \text{dom}(\Gamma)$ and for each $(x : \tau \text{ as } \widehat{\tau}) \in \Gamma$, we have $\Delta, \Gamma \vdash \sigma(x) : \tau$.

We finally state some early results on high-level typing, which will be used for proving type soundness in Section 3.4. In the following, we assume Δ, Γ, σ, $\tau \in \mathit{Types}$ and $\theta \in \mathit{ValuExprs} \cup \mathit{Patterns}$ such that:

$$\vDash \Delta \qquad \Delta \vDash \Gamma \qquad \Delta, \Gamma \vDash \sigma \qquad \Delta, \Gamma \vdash \theta : \tau$$

Lemma 3.1 (Focusing traverses high-level typing)**.** *For any path π, if* $\textbf{focus}\ (\pi, \theta)$ *is defined, then* $\textbf{focus}\ (\pi, \tau)$ *is defined and we have* $\Delta, \Gamma \vdash \textbf{focus}\ (\pi, \theta) : \textbf{focus}\ (\pi, \tau)$.

Proof. Immediate by induction. □

Lemma 3.2 (Accessors and their bound values have the same high-level type)**.** *Let* $x \in \text{dom}(\sigma)$ *and* $\pi \in \mathit{Paths}$. *We have* $\Delta, \Gamma \vdash x.\pi : \tau$ *if and only if* $\textbf{focus}\ (\pi, \sigma(x))$ *is defined and* $\Delta, \Gamma \vdash \textbf{focus}\ (\pi, \sigma(x)) : \tau$.

Proof. Immediate. □

3.2.3.2 Typing for source programs

We can now type *expressions*, which represent full-fledged ribbit programs. By design, each value introduced in our language is associated with both an ADT and its memory layout. Our typing judgment for full expressions therefore checks an expression e against both a high-level type τ and a memory type $\widehat{\tau}$, and we write $\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}$. Its rules are defined in Fig. 3.23. Note that in order to type function calls, Γ now also contains bindings of the form $(f : \tau \text{ as } \widehat{\tau} \to \tau' \text{ as } \widehat{\tau}')$ which indicate that the function bound to $f \in \mathit{FunVars}$ takes an argument of type τ represented as $\widehat{\tau}$, and returns a value of type τ' represented as $\widehat{\tau}'$.

$$\textsc{THLTyVar}\ \frac{(t \mapsto \tau) \in \Delta \qquad \Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}}{\Delta, \Gamma \vdash e : t \text{ as } \widehat{\tau}} \qquad \textsc{TMTyVar}\ \frac{(t \mapsto \widehat{\tau}) \in \Delta \qquad \Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}}{\Delta, \Gamma \vdash e : \tau \text{ as } t}$$

$$\textsc{TPivot}\ \frac{\Delta \vDash \widehat{\tau} \qquad \mathbf{agree}_{\Delta}(\tau, \widehat{\tau}) \qquad \Delta, \Gamma \vdash u : \tau}{\Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau}) : \tau \text{ as } \widehat{\tau}} \qquad \textsc{TLetBind}\ \frac{\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau} \qquad \Delta, \Gamma \cup \{(x : \tau \text{ as } \widehat{\tau})\} \vdash e' : \tau' \text{ as } \widehat{\tau}'}{\Delta, \Gamma \vdash \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = e\ \mathsf{in}\ e' : \tau' \text{ as } \widehat{\tau}'}$$

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

$$\textsc{TMatch}\ \frac{(x : \tau \text{ as } \widehat{\tau}) \in \Gamma \qquad \Delta \vdash p_i : \tau \qquad \Delta, \Gamma \vdash e_i : \tau' \text{ as } \widehat{\tau}' \qquad \{p_0, \ldots, p_{n-1}\} \text{ is exhaustive for } \tau}{\Delta, \Gamma \vdash \mathsf{match}(x)\{p_0 \to e_0\ \ldots\ p_{n-1} \to e_{n-1}\} : \tau' \text{ as } \widehat{\tau}'}$$

Figure 3.23: Typing judgment for full expressions.

Pivot expressions are precisely where memory types are introduced in an expression. The TPivot rule ensures that the high-level and memory types of every pivot agree with each other, and that all memory types are well-kinded. The following immediate result propagates this property to every well-typed expression:

Lemma 3.3. *If* $\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}$*, then* $\Delta \vDash \widehat{\tau}$ *and* $\mathbf{agree}_{\Delta}(\tau, \widehat{\tau})$.

Proof. Immediate by induction on e. □

The TMatch rule types a pattern matching expression by checking that the left-hand side pattern and right-hand side expression of each branch are well-typed. It also checks pattern matching *exhaustivity*: if the patterns are all of type τ, then every possible value of type τ must be *matched* by at least one of these patterns, using the pattern matching evaluation judgment defined in Fig. 3.26. This is similar to, for instance, the OCaml compiler, in which pattern matching exhaustivity is checked during typing and the pattern is completed if necessary. The actual procedure for checking exhaustivity in our TMatch rule is left unspecified as there is ample literature on the topic (Maranget 2007; Liu 2016).

Example 3.18 (Typing a program on lists)**.** Recall the type variable environment Δ_{list}, the high-level type τ_{list} and the memory type $\widehat{\tau}_p$ from Examples 3.1 and 3.4. Consider the following expression:

$$e = \mathsf{let}\ x : \tau_{\text{list}} \text{ as } \widehat{\tau}_p = \text{Cons}(\langle 42, \text{Nil}\rangle)\ \mathsf{in}\ \mathsf{match}(x) \left\{ \begin{array}{lll} \text{Nil} & \to & (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\_) & \to & (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\}$$

It is immediate from the definitions of validity, kinding and agreement that the memory type I_{32} is valid and well-kinded, and that it agrees with its high-level counterpart I_{32}. Following the same reasoning as Examples 3.11 and 3.12, we show that $\widehat{\tau}_p$ is valid and well-kinded. We have shown that τ_{list} and $\widehat{\tau}_p$ agree in Example 3.13. Using HLTTypeVar, HLTConstr and HLTWildcard rules, we show that both patterns Nil and Cons(_) are of type t_{list}. Using the results from Example 3.17 and with $\Gamma = \{(x : t_{\text{list}} \text{ as } \widehat{\tau}_p)\}$, we

can now type e:

$$\textsc{TPivot}\ \frac{\Delta_{\text{list}} \vDash \widehat{\tau}_p \qquad \mathbf{agree}_{\Delta_{\text{list}}}(\tau_{\text{list}}, \widehat{\tau}_p) \qquad \Delta_{\text{list}}, \varnothing \vdash \text{Cons}(\langle 42, \text{Nil}\rangle) : \tau_{\text{list}}}{\Delta_{\text{list}}, \varnothing \vdash (\text{Cons}(\langle 42, \text{Nil}\rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p) : t_{\text{list}} \text{ as } \widehat{\tau}_p}$$

$$\textsc{TPivot}\ \frac{\Delta_{\text{list}} \vDash I_{32} \qquad \mathbf{agree}_{\Delta_{\text{list}}}(I_{32}, I_{32}) \qquad \Delta_{\text{list}}, \Gamma \vdash x.\text{Cons}.0 : I_{32}}{\Delta_{\text{list}}, \Gamma \vdash (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) : I_{32} \text{ as } I_{32}}$$

$$\textsc{TPivot}\ \frac{\Delta_{\text{list}} \vDash I_{32} \qquad \mathbf{agree}_{\Delta_{\text{list}}}(I_{32}, I_{32}) \qquad \textsc{HLTConstant}\ \dfrac{0 \leqslant 0 < 2^{32}}{\Delta_{\text{list}}, \Gamma \vdash 0 : I_{32}}}{\Delta_{\text{list}}, \Gamma \vdash (0 : I_{32} \text{ as } I_{32}) : I_{32} \text{ as } I_{32}}$$

$$\textsc{TMatch}\ \frac{\begin{array}{c}(x : t_{\text{list}} \text{ as } \widehat{\tau}_p) \in \Gamma \\ \Delta_{\text{list}} \vdash \text{Nil} : t_{\text{list}} \qquad \Delta_{\text{list}} \vdash \text{Cons}(\_) : t_{\text{list}} \qquad \Delta_{\text{list}}, \Gamma \vdash (0 : I_{32} \text{ as } I_{32}) : I_{32} \text{ as } I_{32} \\ \Delta_{\text{list}}, \Gamma \vdash (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) : I_{32} \text{ as } I_{32} \qquad \text{Nil} \rhd \text{Nil} \wedge \forall v, \text{Cons}(\_) \rhd \text{Cons}(v)\end{array}}{\Delta_{\text{list}}, \Gamma \vdash \text{match}(x) \left\{ \begin{array}{lcl} \text{Nil} & \to & (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\_) & \to & (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\} : I_{32} \text{ as } I_{32}}$$

$$\textsc{TLetBind}\ \frac{\begin{array}{c}\Delta_{\text{list}}, \varnothing \vdash (\text{Cons}(\langle 42, \text{Nil}\rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p) : t_{\text{list}} \text{ as } \widehat{\tau}_p \\ \Gamma = \{(x : t_{\text{list}} \text{ as } \widehat{\tau}_p)\} \qquad \Delta_{\text{list}}, \Gamma \vdash \text{match}(x) \left\{ \begin{array}{lcl} \text{Nil} & \to & (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\_) & \to & (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\} : I_{32} \text{ as } I_{32}\end{array}}{\Delta_{\text{list}}, \varnothing \vdash e : I_{32} \text{ as } I_{32}}$$

△

In order to define an evaluation judgment for expressions (in Section 3.3.2), we need a notion of *function environments*, usually denoted Σ, which bind function symbols to lambda-expressions of the form $\lambda x.e$. Such an environment is well-typed iff. each bound expression is well-typed:

Definition 3.4 (well-typed function environments)**.** Let Δ a well-formed type variable environment and Γ a well-formed typing environment in Δ. A function binding environment Σ is *well-typed* in Δ and Γ, and we write $\Delta, \Gamma \vdash \Sigma$, if and only if $\text{dom}(\Sigma) = \text{dom}(\Gamma)$ and for each $(f : \tau \text{ as } \widehat{\tau} \to \tau' \text{ as } \widehat{\tau}') \in \Gamma$ with $\Sigma(f) = \lambda x.e$, we have $\Delta, \Gamma \cup \{x : \tau \text{ as } \widehat{\tau}\} \vdash e : \tau' \text{ as } \widehat{\tau}'$.

3.2.4 Typing for memory-level objects

The last typing judgment we need to define in order to define a semantics for our language and prove its soundness is *memory typing*. This judgment, denoted $\Delta, \Gamma, \varsigma \vdash \widehat{e} : \widehat{\tau}$ and defined in Fig. 3.24, assigns a memory type $\widehat{\tau}$ to a memory expression $\widehat{e}$ in the context of a memory store ς.

As defined in Fig. 3.17, a memory expression $\widehat{e} \in \widehat{\mathit{Exprs}}$ is either a memory valuexpression $\widehat{u} \in \widehat{\mathit{ValuExprs}}$ (whose grammar also covers memory values and patterns), a high-level expression $e \in \mathit{Exprs}$, or an intermediate let-binding form $\text{let } x : \tau \text{ as } \widehat{\tau} = \widehat{e} \text{ in } e$ with $\widehat{e} \notin \mathit{Exprs}$. The two latter forms – high-level expressions and let-bindings – are handled by MemTHLExp and MemTLet respectively, which rely on the previously defined typing judgment for high-level expressions.

Memory valuexpressions require a *store* ς mapping addresses to memory values, in addition to the usual type variable and typing environments. We use it in the MemTAddress rule, to type pointer memory values (of the form $\&_\ell(a)$) which, unlike pointer expressions (of the form $\&_\ell(\widehat{u})$), do not embed the memory value they point to and instead only contain its address. Most of the other rules are straightforward, defined by induction on $\widehat{\tau}$. Note that we ignore the parts of memory types that are related to the high-level type they represent: for instance, MemTSplit assigns a split type to any expression which is accepted by the right-hand side of a branch, regardless of whether it actually represents a value of the adequate provenance. Additional constraints on splits and fragments are enforced by agreement criteria (Section 3.2.2) and high-level typing judgments (Section 3.2.3).

$$\textsc{MemTHLExp}\ \frac{\Delta,\Gamma \vdash e : \tau \text{ as } \widehat{\tau}}{\Delta,\Gamma,\varsigma \vdash e : \widehat{\tau}} \qquad \textsc{MemTLet}\ \frac{\Delta,\Gamma,\varsigma \vdash \widehat{e} : \widehat{\tau} \qquad \mathbf{agree}(\tau,\widehat{\tau}) \qquad \Delta,\Gamma \cup \{(x : \tau \text{ as } \widehat{\tau})\} \vdash e : (\tau' \text{ as } \widehat{\tau}')}{\Delta,\Gamma,\varsigma \vdash \text{let } x : \tau \text{ as } \widehat{\tau} = \widehat{e} \text{ in } e : \widehat{\tau}'}$$

$$\textsc{MemTTypeVar}\ \frac{(t \mapsto \widehat{\tau}) \in \Delta \qquad \Delta,\Gamma,\varsigma \vdash \widehat{e} : \widehat{\tau}}{\Delta,\Gamma,\varsigma \vdash \widehat{e} : t} \qquad \textsc{MemTPrimitive}\ \Delta,\Gamma,\varsigma \vdash (c)_\ell : I_\ell$$

$$\textsc{MemTFission}\ \frac{o_0 = 0 \qquad o_{n-1} + \ell_{n-1} = \ell \qquad o_i = o_{i-1} + \ell_{i-1} \qquad \Delta,\Gamma,\varsigma \vdash \widehat{u}_i : I_{\ell_i}}{\Delta,\Gamma,\varsigma \vdash \__\ell \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \widehat{u}_i : I_\ell} \qquad \textsc{MemTWord}\ \Delta,\Gamma,\varsigma \vdash \__\ell : \__\ell$$

$$\textsc{MemTConstant}\ \Delta,\Gamma,\varsigma \vdash (c)_\ell : (c)_\ell \qquad \textsc{MemTAddress}\ \frac{a \notin \mathrm{dom}(\varsigma) \qquad \Delta,\Gamma,\varsigma \vdash \widehat{v} : \widehat{\tau}}{\Delta,\Gamma,\varsigma \cup \{a \mapsto \widehat{v}\} \vdash \&_\ell(a) : \&_\ell(\widehat{\tau})} \qquad \textsc{MemTPointer}\ \frac{\Delta,\Gamma,\varsigma \vdash \widehat{u} : \widehat{\tau}}{\Delta,\Gamma,\varsigma \vdash \&_\ell(\widehat{u}) : \&_\ell(\widehat{\tau})}$$

$$\textsc{MemTComposite}\ \frac{\Delta,\Gamma,\varsigma \vdash \widehat{u} : \widehat{\tau} \qquad \Delta,\Gamma,\varsigma \vdash \widehat{u}_i : \widehat{\tau}_i}{\Delta,\Gamma,\varsigma \vdash \widehat{u} \bowtie_{0 \leqslant i < n} r_i : \widehat{u}_i : \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i} \qquad \textsc{MemTStruct}\ \frac{\Delta,\Gamma,\varsigma \vdash \widehat{u}_i : \widehat{\tau}_i}{\Delta,\Gamma,\varsigma \vdash \{\!\{\widehat{u}_0,\ldots,\widehat{u}_{n-1}\}\!\} : \{\!\{\widehat{\tau}_0,\ldots,\widehat{\tau}_{n-1}\}\!\}}$$

$$\textsc{MemTFragment}\ \frac{\Delta,\Gamma,\varsigma \vdash \widehat{e} : \widehat{\tau}}{\Delta,\Gamma,\varsigma \vdash \widehat{e} : (\pi \text{ as } \widehat{\tau})} \qquad \textsc{MemTSplit}\ \frac{\widehat{\tau} = \mathrm{split}(\ldots) \qquad \exists (p,\widehat{\tau}') \in \widehat{\tau}/\_ \qquad \Delta,\Gamma,\varsigma \vdash \widehat{e} : \widehat{\tau}'}{\Delta,\Gamma,\varsigma \vdash \widehat{e} : \widehat{\tau}}$$

Figure 3.24: Memory-level typing judgment. Environments appearing in gray in a rule are irrelevant to its application.

Example 3.19 (Typing list memory values and expressions)**.** Recall the memory type $\widehat{\tau}_p$ and the memory value $\widehat{v}$ from Example 3.7:

$$\begin{aligned}
&\widehat{\tau}_p = \mathrm{split}(.[0:2])\,\{\\
&\quad 0 \text{ from } \mathrm{Nil} &&\Rightarrow __{64} \ltimes [0:2] : (0)_2\\
&\quad 1 \text{ from } \mathrm{Cons}(\langle _, \mathrm{Nil}\rangle) &&\Rightarrow __{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } I_{32})\\
&\quad 2 \text{ from } \mathrm{Cons}(\langle _, \mathrm{Cons}(_)\rangle) &&\Rightarrow \&_{64}\left(\{\!\{(.\mathrm{Cons}.0 \text{ as } I_{32}), (.\mathrm{Cons}.1.\mathrm{Cons}.0 \text{ as } I_{32}), (.\mathrm{Cons}.1.\mathrm{Cons}.1 \text{ as } t_p)\}\!\}\right)\\
&&&\quad \ltimes [0:2] : (2)_2\\
&\}
\end{aligned}$$

$$\widehat{v} = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

We show that $\widehat{v}$ is of type $\widehat{\tau}_p$:

$$\textsc{MemTSplit}\ \frac{\textsc{MemTComposite}\ \dfrac{\textsc{MemTWord}\ \dfrac{}{\Delta_{list} \vdash \__{64} : \__{64}} \qquad \textsc{MemTConstant}\ \dfrac{}{\Delta_{list} \vdash (1)_2 : (1)_2} \qquad \textsc{MemTFragment}\ \dfrac{\textsc{MemTPrimitive}\ \dfrac{}{\Delta_{list} \vdash (42)_{32} : I_{32}}}{\Delta_{list} \vdash (42)_{32} : (.\mathrm{Cons}.0 \text{ as } I_{32})}}{\Delta_{list} \vdash \widehat{v} : \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } I_{32})} \qquad (\mathrm{Cons}(\langle \_, \mathrm{Nil}\rangle), \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } I_{32})) \in \widehat{\tau}_p/\_}{\Delta_{list} \vdash \widehat{v} : \widehat{\tau}_p}$$

Using results from Example 3.18 and the MemTLet rule, we can then show that the memory expression from Example 3.8 $\widehat{e} = \text{let } x : \tau_{list} \text{ as } \widehat{\tau}_p = \widehat{v} \text{ in } (x.\mathrm{Cons}.0 : I_{32} \text{ as } I_{32})$ is of type I_{32}. △

The MemTFission rule is slightly unusual, in that it assigns a primitive type I_ℓ to a composite memory value. Its purpose is to allow the interpretation of "mangled" integer values – that is, words which are entirely filled with integers on disjoint bit ranges – as integers, even though their shape (composite word) does not immediately match that of a primitive type.

Example 3.20 (Mangled integer value)**.** Consider the high-level primitive type of 64-bit integers I_{64}, together with the following memory type which splits it into two 32-bit pieces:

$$\_{}_{64} \ltimes [0:32] : (.[0:32] \text{ as } \mathrm{I}_{32}) \ltimes [32:32] : (.[32:32] \text{ as } \mathrm{I}_{32})$$

Using this layout, the high-level value `0x111100002222` is represented as the following memory value:

$$\_{}_{64} \ltimes [0:32] : (\texttt{0x2222})_{32} \ltimes [32:32] : (\texttt{0x1111})_{32}$$

The TFission typing rule lets us assign the type I_{64} to this value to reinterpret it as the direct 64-bit integer encoding $(\texttt{0x111100002222})_{64}$. △

In practice, this rule lets us capture memory layouts that split primitive values into multiple pieces scattered across the memory type, such as the RISC-V layout presented in Section 2.5.

Now that we have defined well-typed memory values and expressions, we can extend this judgment to their environments. We define well-formed typing environments which, similar to well-formed type variable environments, apply agreement criteria to every bound type pair.

Definition 3.5 (well-formed typing environments)**.** Let Δ a well-formed type variable environment. A typing environment Γ is well-formed in Δ, and we write $\Delta \vDash \Gamma$, if and only if:

- for each $(x : \tau \text{ as } \widehat{\tau}) \in \Gamma$, we have $\Delta \vDash \widehat{\tau}$ and $\mathbf{agree}_\Delta(\tau, \widehat{\tau})$;
- for each $(f : \tau \text{ as } \widehat{\tau} \to \tau' \text{ as } \widehat{\tau}') \in \Gamma$, we have $\Delta \vDash \widehat{\tau}$, $\mathbf{agree}_\Delta(\tau, \widehat{\tau})$, $\Delta \vDash \widehat{\tau}'$ and $\mathbf{agree}_\Delta(\tau', \widehat{\tau}')$.

Similar to high-level variable binding environments, a *memory value binding environment* $\widehat{\sigma} : \mathit{Vars} \to \widehat{\mathit{Values}}$ is well-typed iff. every bound value is well-typed.

Definition 3.6 (well-typed memory environments)**.** Let Δ a well-formed type declaration environment, Γ a well-formed typing environment in Δ and ς a memory store. A memory value binding environment $\widehat{\sigma}$ is *well-typed* in Δ, Γ and ς, and we write $\Delta, \Gamma, \varsigma \vdash \widehat{\sigma}$, if and only if $\mathrm{dom}(\widehat{\sigma}) = \mathrm{dom}(\Gamma)$ and for each $(x : \tau \text{ as } \widehat{\tau}) \in \Gamma$, we have $\Delta, \Gamma, \varsigma \vdash \widehat{\sigma}(x) : \widehat{\tau}$.

We finally state an immediate result on memory typing which will be used for proving type soundness in Section 3.4.

Lemma 3.4 (memory focusing traverses memory typing)**.** *Let Δ a well-formed type declaration environment, Γ a well-formed typing environment in Δ and ς a memory store. Let $\widehat{v} \in \widehat{\mathit{Values}}$ and $\widehat{\tau} \in \widehat{\mathit{Types}}$ such that $\Delta, \Gamma, \varsigma \vdash \widehat{v} : \widehat{\tau}$. For all $\widehat{\pi} \in \widehat{\mathit{Paths}}$ such that $\widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau})$ is defined, we have $\Delta, \Gamma, \varsigma \vdash \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \widehat{v}) : \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{\tau})$.*

Proof. Immediate by induction. □

3.3 Semantics

Now that syntactical constructs and their typing and validity judgments have been formalized, we are finally able to define a two-tiered operational semantics for our language that takes both high-level and memory constructs into account.

3.3.1 High-level expression evaluation

The small-step evaluation judgment for high-level programs, defined in Fig. 3.25 and denoted $\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', e'$, operates on triples consisting of a typing environment Γ, a binding environment σ mapping variables to valuexpressions, and an expression $e \in \mathit{Exprs}$. It is deterministic and its normal forms are pivot expressions of the form $(u : \tau \text{ as } \widehat{\tau})$. The function environment Σ maps function names $f \in \mathit{FunVars}$ to lambda-expressions of the form $\lambda x.e$. We assume that variables have been renamed in e so that they are bound at most once, regardless of scope.

$$\textsc{HLEFunCall}\quad \frac{(x : \tau \text{ as } \widehat{\tau}) \in \Gamma \qquad (f : \tau \text{ as } \widehat{\tau} \to \tau' \text{ as } \widehat{\tau'}) \in \Gamma \qquad (x \mapsto u) \in \sigma \qquad (f \mapsto \lambda x'.e) \in \Sigma}{\Sigma \vdash \Gamma, \sigma, f(x) \hookrightarrow \Gamma \cup \{x' : \tau \text{ as } \widehat{\tau}\}, \sigma \cup \{x' \mapsto u\}, e}$$

$$\textsc{HLELetStep}\quad \frac{\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', e'}{\Sigma \vdash \Gamma, \sigma, \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = e\ \mathsf{in}\ e_0 \hookrightarrow \Gamma', \sigma', \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = e'\ \mathsf{in}\ e_0}$$

$$\textsc{HLELetBind}\quad \Sigma \vdash \Gamma, \sigma, \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = (u : \tau' \text{ as } \widehat{\tau'})\ \mathsf{in}\ e \hookrightarrow \Gamma \cup \{x : \tau \text{ as } \widehat{\tau}\}, \sigma \cup \{x \mapsto u\}, e$$

$$\textsc{HLEMatch}\quad \frac{x \in \mathrm{dom}(\sigma) \qquad \exists i, \sigma \vdash p_i \rhd \sigma(x) \qquad \forall j < i, \sigma \vdash p_j \not\rhd \sigma(x)}{\Sigma \vdash \Gamma, \sigma, \mathsf{match}(x)\{p_1 \to e_1\ \ldots\ p_n \to e_n\} \hookrightarrow \Gamma, \sigma, e_i}$$

Figure 3.25: High-level expression evaluation.

Note that reducing an expression with $\hookrightarrow$ until a normal form is reached does not yield a value, but a (typed) valuexpression which may be reduced further by applying substitutions from the binding environment σ. These two layers of evaluation are separated so that it is possible to establish a correspondence between high-level and memory-level evaluation, shown in Section 3.3.2.

The HLEMatch rule relies on an ancillary big-step pattern matching judgment, denoted $\rhd$ and defined in Fig. 3.26. Given a pattern p, a valuexpression or pattern θ of the same type and a binding environment σ, we write $\sigma \vdash p \rhd \theta$ if p *matches* θ using σ to substitute variables, and $\sigma \vdash p \not\rhd \theta$ otherwise. σ may be omitted if θ is a value or a pattern.

$$\sigma \vdash \_ \rhd \theta \qquad \frac{\sigma \vdash p \rhd \mathbf{focus}\ (\pi, v)}{\sigma \cup \{x \mapsto v\} \vdash p \rhd x.\pi} \qquad \sigma \vdash c \rhd c \qquad \frac{\sigma \vdash p_i \rhd \theta_i}{\sigma \vdash \langle p_0, \ldots, p_{n-1}\rangle \rhd \langle \theta_0, \ldots, \theta_{n-1}\rangle} \qquad \frac{\sigma \vdash p \rhd \theta}{\sigma \vdash K(p) \rhd K(\theta)}$$

Figure 3.26: High-level pattern matching judgment with $\theta \in \mathit{ValuExprs} \cup \mathit{Patterns}$ and a binding environment σ

Example 3.21 (High-level evaluation with lists)**.** Consider the following expression, taken from Example 3.18:

$$\mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_p = \mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle)\ \mathsf{in}\ \mathsf{match}(x) \left\{ \begin{array}{ll} \mathrm{Nil} & \to (0 : I_{32} \text{ as } I_{32}) \\ \mathrm{Cons}(\_) & \to (x.\mathrm{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\}$$

It creates a value of type τ_{list} represented using the $\widehat{\tau}_p$ layout defined in Example 3.4. It then matches it against patterns and extracts its first element. We reduce it to a pivot expression with the two following high-level evaluation steps. We omit the empty function environment, merge typing and binding environments, and indicate which elements are affected by a reduction step by highlighting them in the

corresponding color.

$$\sigma = \varnothing,\ \text{let } x : \tau_{\text{list}} \text{ as } \widehat{\tau}_p = \text{Cons}(\langle 42, \text{Nil}\rangle) \text{ in } \text{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\_) & \to (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\}$$

$\hookrightarrow$(HLELetBind)

$$\sigma = \{\ x : \tau_{\text{list}} \text{ as } \widehat{\tau}_p \mapsto \text{Cons}\ (\langle 42, \text{Nil}\rangle)\ \},\ \text{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to (0 : I_{32} \text{ as } I_{32}) \\ \text{Cons}(\_) & \to (x.\text{Cons}.0 : I_{32} \text{ as } I_{32}) \end{array} \right\}$$

$\hookrightarrow$(HLEMatch)

$$\sigma = \{x : \tau_{\text{list}} \text{ as } \widehat{\tau}_p \mapsto \text{Cons}(\langle 42, \text{Nil}\rangle)\},\ (x.\text{Cons}.0 : I_{32} \text{ as } I_{32})$$

The second step (HLEMatch) leads to the expression on the right-hand-side of the Cons(_) pattern matching branch, since we have $\text{Nil} \not\rhd \text{Cons}(\langle 42, \text{Nil}\rangle)$ and $\text{Cons}(\_) \rhd \text{Cons}(\langle 42, \text{Nil}\rangle)$. △

We now state and prove the soundness of our semantics w.r.t. the high-level typing judgment defined in Section 3.2.3.

Theorem 3.1 (high-level type soundness). *Let $\Delta, \Sigma, \Gamma, \sigma, \tau, \widehat{\tau}$ and e such that:*

$$\vDash \Delta \qquad \Delta \vDash \Gamma \qquad \Delta, \Gamma \vdash \Sigma \qquad \Delta, \Gamma \vdash \sigma \qquad \Delta \vDash \widehat{\tau} \qquad \textbf{\textit{agree}}_{\Delta}(\tau, \widehat{\tau}) \qquad \Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}$$

We have the two following properties:

Preservation: *for all Γ', σ', e' such that $\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', e'$, we have:*

$$\Delta \vDash \Gamma' \qquad \Delta, \Gamma' \vdash \Sigma \qquad \Delta, \Gamma' \vdash \sigma' \qquad \Delta, \Gamma' \vdash e' : \tau \text{ as } \widehat{\tau}$$

Progress: *either e is a pivot expression (normal form) or there exist Γ', σ' and e' such that $\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', e'$.*

Proof. By induction on $\Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}$.

- If τ or $\widehat{\tau}$ is a type variable t, we refer to the case corresponding to $\Delta(t)$.
- If $e = (u : \tau \text{ as } \widehat{\tau})$, it is a normal form w.r.t. $\hookrightarrow$.
- If $e = \text{let } x = e' \text{ in } e_0$, there exist τ' and $\widehat{\tau}'$ such that $\textbf{agree}(\tau', \widehat{\tau}')$, $\Delta, \Gamma \vdash e' : \tau' \text{ as } \widehat{\tau}'$ and $\Delta, \Gamma_0 \vdash e_0 : \tau \text{ as } \widehat{\tau}$ where $\Gamma_0 = \Gamma \cup \{(x : \tau' \text{ as } \widehat{\tau}')\}$.
  - If $e' = (u : \tau' \text{ as } \widehat{\tau}')$, we have $\textbf{agree}(\tau', \widehat{\tau}')$ and $\Delta, \Gamma \vdash u : \tau'$. Let $\sigma_0 = \sigma \cup \{x \mapsto u\}$. We have $\Delta, \Gamma_0 \vdash \sigma_0$ and exactly one evaluation rule applies: $\Gamma, \sigma, e \hookrightarrow \Gamma_0, \sigma_0, e_0$.
  - Otherwise, we use the induction hypothesis for Γ, σ and e'.

    Progress: since e' is not in normal form, there exist Γ', σ' and e'' such that $\Sigma \vdash \Gamma, \sigma, e' \hookrightarrow \Gamma', \sigma', e''$ and we have $\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', \text{let } x = e'' \text{ in } e_0$.

    Preservation: the only applicable evaluation rules are of the form $\Sigma \vdash \Gamma, \sigma, e \hookrightarrow \Gamma', \sigma', \text{let } x = e'' \text{ in } e_0$ with $\Sigma \vdash \Gamma, \sigma, e' \hookrightarrow \Gamma', \sigma', e''$. We have $\Delta \vDash \Gamma'$, $\Delta, \Gamma' \vdash \Sigma$, $\Delta, \Gamma' \vdash \sigma'$ and $\Delta, \Gamma' \vdash e'' : \tau' \text{ as } \widehat{\tau}'$. Let $\Gamma'_0 = \Gamma' \cup \{(x : \tau' \text{ as } \widehat{\tau}')\}$. Assuming that all variable symbols are unique, since we never remove bindings from Γ, we have $\Delta, \Gamma'_0 \vdash e_0 : \tau \text{ as } \widehat{\tau}$, hence $\Delta, \Gamma' \vdash \text{let } x = e'' \text{ in } e_0 : \tau \text{ as } \widehat{\tau}$.
- If $e = f(x)$, there exist τ_x and $\widehat{\tau}_x$ such that $\textbf{agree}(\tau_x, \widehat{\tau}_x)$, $(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma$ and $(f : (\tau_x \text{ as } \widehat{\tau}_x) \to (\tau \text{ as } \widehat{\tau})) \in \Gamma$. Let $(\lambda x'.e') = \Sigma(f)$, $\Gamma' = \Gamma \cup \{(x' : \tau_x \text{ as } \widehat{\tau}_x)\}$ and $\sigma' = \sigma \cup \{x' \mapsto \sigma(x)\}$. We have (well-typed environments) $\Delta, \Gamma' \vdash e' : \tau \text{ as } \widehat{\tau}$, $\Delta, \Gamma \vdash \sigma(x) : \tau(x)$, hence $\Delta, \Gamma' \vdash \sigma'$, and exactly one evaluation rule applies: $\Sigma \vdash \Gamma, \sigma, f(x) \hookrightarrow \Gamma', \sigma', e'$.
- If $e = \text{match}(x)\left\{ p_i \to e_i \to \middle| \ 0 < i \leqslant n \right\}$, there exist τ_x and $\widehat{\tau}_x$ such that $\textbf{agree}(\tau_x, \widehat{\tau}_x)$, $(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma$, $\Delta, \Gamma \vdash p_i : \tau_x$ and $\Delta, \Gamma \vdash e_i : \tau \text{ as } \widehat{\tau}$ for each i, and at least one pattern matches $\sigma(x)$. Let $i = \text{argmin}\{p_i \rhd \sigma(x)\}$. Exactly one evaluation rule applies: $\Gamma, \sigma, e \hookrightarrow \Gamma, \sigma, e_i$ and we have $\Delta, \Gamma \vdash e_i : \tau \text{ as } \widehat{\tau}$.

□

3.3.2 Memory-level evaluation

The evaluation judgment defined in the previous section reduces all high-level language constructs, but completely ignores memory-related elements, stopping at pivot expressions. We now formalize a *memory-level* semantics for our language, which handles high-level constructs (matches, let-bindings and function calls), but also reduces pivot expressions to memory values. By doing so, we also define which memory values "properly represent" a high-level value according to a given memory layout – they correspond to the result of fully evaluating this initial pivot expression. For instance, we reduce the following pivot expression to its representation as a memory value in 6 memory-level steps:

$$(\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}}) \rightsquigarrow_{\mathrm{m}}^{6} \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

Let us first informally introduce some key evaluation steps on our running example.

Example 3.22 (Memory-level evaluation on lists)**.** Consider the following memory-level evaluation sequence, which reduces the expression e from Example 3.7 to a memory value:

$$\begin{array}{l}
\quad \varnothing, \mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} = (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}})\ \mathsf{in}\ \mathsf{match}(x) \left\{ \begin{array}{lll} \mathrm{Nil} & \to & (0 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}) \\ \mathrm{Cons}(_) & \to & (x.\mathrm{Cons}.0 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}) \end{array} \right\} \\
\rightsquigarrow_{\mathrm{m}} (\textsc{ESplit}) \\
\quad \mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} = \\
\quad \varnothing, \quad (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Nil}\rangle) \text{ as } _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})) \\
\quad \mathsf{in}\ \mathsf{match}(x)\{\ldots\} \\
\rightsquigarrow_{\mathrm{m}}^{3} (\textsc{EComposite}, \textsc{EWord}, \textsc{EConstant}) \\
\quad \mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} = \\
\quad \varnothing, \quad _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Nil}\rangle) \text{ as } (.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})) \\
\quad \mathsf{in}\ \mathsf{match}(x)\{\ldots\} \\
\rightsquigarrow_{\mathrm{m}} (\textsc{EFragment}) \\
\quad \varnothing, \mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} = _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32})\ \mathsf{in}\ \mathsf{match}(x)\{\ldots\} \\
\rightsquigarrow_{\mathrm{m}} (\textsc{EAtom}) \\
\quad \varnothing, \mathsf{let}\ x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} = _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\ \mathsf{in}\ \mathsf{match}(x)\{\ldots\} \\
\left.\begin{array}{l}
\rightsquigarrow_{\mathrm{h}} (\textsc{ELetBind}) \\
\quad \{x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} \mapsto _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\}, \mathsf{match}(x) \left\{ \begin{array}{lll} \mathrm{Nil} & \to & (0 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}) \\ \mathrm{Cons}(_) & \to & (x.\mathrm{Cons}.0 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}) \end{array} \right\} \\
\rightsquigarrow_{\mathrm{h}} (\textsc{EMatch}) \\
\quad \{x : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}} \mapsto _{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\}, (x.\mathrm{Cons}.0 : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32})
\end{array}\right\} \text{high-level steps} \\
\rightsquigarrow_{\mathrm{m}} (\textsc{EVarFocus}) \\
\quad \{x : \ldots, x' : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32} \mapsto (42)_{32}\}, (x'.\varepsilon : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}) \\
\rightsquigarrow_{\mathrm{m}} (\textsc{EVarAccess}) \\
\quad \{x : \ldots, x' : \ldots\}, (42)_{32}
\end{array}$$

Our memory-level evaluation judgment, denoted $\rightsquigarrow$, defines a traditional call-by-value semantics for high-level constructs. For instance, we evaluate let-binding expressions by first reducing the let-bound expression to a normal form, then binding it in a value environment and continuing evaluation. However, unlike $\hookrightarrow$, the normal forms of $\rightsquigarrow$ are memory values, rather than pivot expressions. Pivot expressions are precisely where their behaviors begin to diverge. The $\rightsquigarrow$ reduction steps from a pivot expression to a memory value are local to this particular expression; we temporarily "forget" about high-level constructs and instead create concrete memory structures, using the memory type as a guide, until a memory value is reached. We label these memory-level evaluation steps with an m. On this example, we first evaluate the let-bound pivot expression $(\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_{\mathrm{p}})$. Its informal meaning is "produce a memory value that represents the high-level value $\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle)$ (which is of type τ_{list}) according to the memory layout $\widehat{\tau}_{\mathrm{p}}$". Its evaluation is driven by its memory type $\widehat{\tau}_{\mathrm{p}}$ and yields the memory value $\_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$ in six $\rightsquigarrow_{\mathrm{m}}$ steps (ESplit, EComposite, EWord, EConstant, EFragment, EAtom) which we will detail in the rest of this section.

Once the expression is fully reduced, $\rightsquigarrow$ resumes evaluation of the surrounding high-level constructs, going back and forth between memory- and high-level evaluation until a single memory value is reached. On our example, we bind the previously computed memory value to x in the environment with a ELetBind step, then evaluate the pattern matching expression by selecting the appropriate branch with a EMatch step. Both of these rules have a direct counterpart in the high-level semantics $\hookrightarrow$. We label these high-level reduction steps with an h.

Finally we reach the pivot expression (x.Cons.0 : I_{32} as I_{32}). Once again, we must reduce it to a memory value. Unlike the previous pivot expression, which contained a high-level value (Cons($\langle 42, \text{Nil}\rangle$)), this pivot contains an *accessor* x.Cons.0 instead. Its informal semantics is "retrieve relevant data from the memory representation of x to encode its subterm at position .Cons.0 as a 32-bit integer". As hinted in Section 2.5, even though this task is trivial in many situations, such accessors can present significant challenges with some combinations of layouts. Here, we must bind an intermediate value x′ and perform two $\rightsquigarrow_m$ steps to get the desired piece of data. The final result is the memory value $(42)_{32}$.

Intuitively, we can see that this reduction strategy is "coherent" with the result of the $\hookrightarrow$ evaluation sequence from Example 3.21, which is the pivot expression (x.Cons.0 : I_{32} as I_{32}) with x bound to Cons($\langle 42, \text{Nil}\rangle$): $(42)_{32}$ is indeed the memory representation of **focus** (.Cons.0, Cons($\langle 42, \text{Nil}\rangle$)) = 42 according to the memory layout I_{32}. Section 3.4 will formalize and prove this equivalence between high-level and memory evaluation results. △

A memory-level evaluation state consists of a type environment Γ, a memory value environment $\widehat{\sigma}$, a memory store ς and a memory expression to evaluate $\widehat{e}$. We denote an evaluation step in the type variable environment Δ and function environment Σ with: $\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \widehat{e} \rightsquigarrow \Gamma', \widehat{\sigma}', \varsigma', \widehat{e}'$. When Δ and Σ are immediate from the context, they may be omitted from the judgment.

As shown in Section 3.1, memory expressions include both high-level constructs – pattern matching, let-bindings and function calls – and memory-level structures. A "full" $\rightsquigarrow$ evaluation sequence reduces a high-level expression to a memory value; memory expressions capture all intermediate stages that may appear during this process.

The full evaluation judgment $\rightsquigarrow$ is the union of $\rightsquigarrow_h$, which handles high-level constructs in $\widehat{e}$, and $\rightsquigarrow_m$, which follows memory types to create concrete memory structures. Its normal forms are memory values. $\rightsquigarrow_h$ rules, defined in Fig. 3.27, are similar to $\hookrightarrow$ and reduce arbitrary expressions to pivot expressions, while $\rightsquigarrow_m$ rules, defined in Fig. 3.30, reduce pivot expressions to memory values. In both definitions, environments appear in gray in a rule when they are unchanged and unused by this rule. Note that the semantics defined by $\rightsquigarrow$ is *not* equivalent to the sequence of $\hookrightarrow$ and $\rightsquigarrow_m$; high-level ($\rightsquigarrow_h$) steps may be interleaved with memory-level ($\rightsquigarrow_m$) steps (on different subexpressions). While defining $\rightsquigarrow$ as the sequence of $\hookrightarrow$ and $\rightsquigarrow_m$ is possible, this behavior is not coherent with that of the compiled program, which we describe in Chapter 5. Furthermore, $\rightsquigarrow_m$ is non-deterministic, so that it is flexible enough to easily match the behavior of compiled programs.

3.3.2.1 From high-level constructs to pivot expressions

The subset of $\rightsquigarrow$ handling high-level constructs, denoted $\rightsquigarrow_h$ and defined in Fig. 3.27, is mostly similar to the high-level evaluation judgment $\hookrightarrow$. The only major difference between the two is pattern matching evaluation. In $\hookrightarrow$, variables are bound to high-level valuexpressions, and we use the high-level pattern matching judgment $\triangleright$ to determine whether a given pattern matches a high-level valuexpression. However, in $\rightsquigarrow$, variables are bound to *memory values*, which cannot be directly compared to high-level patterns.

The rule EMatch relies a notion of *memory-level pattern matching*, based on memory patterns as defined in Section 3.1. The main idea is to first process each high-level pattern into a list of *memory patterns* matching exactly those memory values that properly represent a high-level value matched by the high-level pattern. We then use a memory-level big-step pattern matching judgment to determine whether one of these memory patterns matches the memory value under scrutiny.

Let us first focus on the **pat2mem** function defined in Fig. 3.28, which lowers a high-level pattern to an equivalent list of memory patterns according to a given memory layout [3].

[3] **pat2mem** is also the first component of pattern matching compilation; see Chapter 4.

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

$$\textsc{ELetBind}\quad \Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \subsetneq, \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = \widehat{v}\ \mathsf{in}\ e \hookrightarrow_h \Gamma \cup \{x : \tau\ \mathsf{as}\ \widehat{\tau}\}, \widehat{\sigma} \cup \{x \mapsto \widehat{v}\}, \subsetneq, e$$

$$\textsc{EFunCall}\quad \frac{(x \mapsto \widehat{v}) \in \widehat{\sigma} \qquad (f \mapsto \lambda x'.e) \in \Sigma}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \subsetneq, f(x) \hookrightarrow_h \Gamma \cup \{x' : \Gamma(x)\}, \widehat{\sigma} \cup \{x' \mapsto \widehat{v}\}, \subsetneq, e}$$

$$\textsc{EMatch}\quad \frac{(x : \tau\ \mathsf{as}\ \widehat{\tau}) \in \Gamma \qquad \exists i, \exists (p, \widehat{p}) \in \mathbf{pat2mem}_\Delta(\widehat{\tau}, p_i).\ \varsigma \vdash \widehat{p} \blacktriangleright \widehat{\sigma}(x) \qquad \forall j < i, \forall (p', \widehat{p}') \in \mathbf{pat2mem}_\Delta(\widehat{\tau}, p_j), \varsigma \vdash \widehat{p}' \not\blacktriangleright \widehat{\sigma}(x)}{\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma, \mathsf{match}(x)\{p_1 \to e_1\ \ldots\ p_n \to e_n\} \hookrightarrow_h \Gamma, \widehat{\sigma}, \varsigma, e_i}$$

Figure 3.27: Memory-level expression evaluation, high-level constructs $\hookrightarrow_h$. Environments appearing in gray in a rule are irrelevant to its application.

$$\begin{aligned}
&\mathbf{pat2mem}_\Delta\{\\
&\quad _\ ,\ \widehat{\tau} && \longrightarrow \{(_, _^{|\widehat{\tau}|})\}\\
&\quad c\ ,\ I_\ell && \longrightarrow \{(c, (c)_\ell)\}\\
&\quad p\ ,\ t && \longrightarrow \mathbf{pat2mem}_\Delta(p, \Delta(t))\\
&\quad p\ ,\ __\ell && \longrightarrow \{(p, __\ell)\}\\
&\quad p\ ,\ (c)_\ell && \longrightarrow \{(p, (c)_\ell)\}\\
&\quad p\ ,\ \&_\ell\,(\widehat{\tau}) && \longrightarrow \left\{(p', \&_\ell\,(\widehat{p})) \;\middle|\; (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau})\right\}\\
&\quad p\ ,\ \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i && \longrightarrow \left\{(p'', \widehat{p} \bowtie_{0 \leqslant i < n} r_i : \widehat{p}_i) \;\middle|\; \begin{array}{l}(p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau})\\ (p_i, \widehat{p}_i) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}_i)\\ p'' = p' \sqcap p_0 \sqcap \ldots \sqcap p_{n-1}\end{array}\right\}\\
&\quad p\ ,\ \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} && \longrightarrow \left\{(p', \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}) \;\middle|\; \begin{array}{l}(p_i, \widehat{p}_i) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}_i)\\ p' = p_0 \sqcap \ldots \sqcap p_{n-1}\end{array}\right\}\\
&\quad p\ ,\ (\pi\ \mathsf{as}\ \widehat{\tau}) && \longrightarrow \left\{(p\,[.\pi \leftarrow p'], \widehat{p}) \;\middle|\; (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(\mathbf{focus}\,(\pi, p), \widehat{\tau})\right\}\\
&\quad p\ ,\ \widehat{\tau} = \mathsf{split}(\ldots) && \longrightarrow \bigcup \left\{\mathbf{pat2mem}_\Delta(p', \widehat{\tau}_b) \;\middle|\; \begin{array}{l}(p_b, \widehat{\tau}_b) \in \widehat{\tau}/_\\ p' = p \sqcap p_b\end{array}\right\}\\
&\}
\end{aligned}$$

Figure 3.28: From high-level to memory patterns, using the type variable environment Δ: **pat2mem**

Given a high-level pattern p and a memory type $\widehat{\tau}$ in the type variable environment Δ, $\mathbf{pat2mem}_\Delta(\widehat{\tau}, p)$ produces a set of branches $(p', \widehat{p})$ consisting of a *refined* high-level pattern p' and of its equivalent memory pattern $\widehat{p}$. The goal is to decompose the pattern into finer (memory) branches. Each branch characterizes a subset of values matched by p by the shape of their memory representation according to $\widehat{\tau}$. Informally, **pat2mem** satisfies the following specification: for any high-level value v matched by p, Given $(p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(\widehat{\tau}, p)$, p' is a more precise version of p and there exists exactly one branch $(p', \widehat{p})$ such that p' matches v and $\widehat{p}$ matches its memory representation according to $\widehat{\tau}$. We state and prove the correctness of **pat2mem** w.r.t. this specification in Section 3.4. More precisely

- If p is a wildcard pattern, or $\widehat{\tau}$ is a constant or empty word type, all values should be accepted. We return a single branch $(p, \widehat{p})$ where $\widehat{p}$ matches all memory values of type $\widehat{\tau}$.
- Primitives and fragments are straightforward by replicating their intended semantics. Exactly one memory pattern matches memory values of type I_ℓ that represents c: $(c)_\ell$. For a fragment $(\pi\ \mathsf{as}\ \widehat{\tau})$,

we capture memory values representing the subterm at $.\pi$ of a value matched by p according to $\widehat{\tau}$.

- Struct and composite word types aggregate multiple fields together. We recursively explore each of these fields, yielding a list of branches for each of them. Possible shapes for the memory values we want to capture correspond to a struct or composite word of the same general shape, with the same number of fields, in which each field belongs to its branch *but also all branches in this combination must be compatible with each other*. To this end, we use pattern intersection and only keep branches for which the intersection between all fields' refined patterns is defined.
- Splits are where p may be forked into multiple subpatterns. Indeed, a high-level pattern may match valuexpressions whose provenances are incompatible – for instance, Cons(_) matches both Cons(⟨x, Nil⟩) and Cons(⟨x, Cons(_)⟩). In this case, we must explore all branches of the split whose provenance set contains at least one provenance matched by p, yielding multiple incompatible branches/refined patterns. Actually, we process all splits at once in the definition: we specialize $\widehat{\tau}$ according to p in which constants have been replaced with wildcards, to obtain provenances, and thus branches.

Example 3.23 (Memory patterns for lists)**.** Our running example features two patterns of type τ_{list}: Nil and Cons(_). According to the memory layout $\widehat{\tau}_p$, Nil translates to a single memory pattern, while Cons(_) yields two memory patterns corresponding to the two branches Cons(⟨_, Nil⟩) and Cons(⟨_, Cons(_)⟩) in the toplevel split. In Example 3.22, we first compute the following patterns to evaluate the pattern matching construct:

$$\textbf{pat2mem}_{\Delta_{\text{list}}}(\text{Nil}, \widehat{\tau}_p) = \{(\text{Nil}, \_{64} \ltimes [0:2] : (0)_2)\}$$

$$\textbf{pat2mem}_{\Delta_{\text{list}}}(\text{Cons}(\_), \widehat{\tau}_p) = \left\{ \begin{array}{ll} (\text{Cons}(\langle \_, \text{Nil}\rangle) & , \ \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : \_{32}) \\ (\text{Cons}(\langle \_, \text{Cons}(\_)\rangle) & , \ \&_{64}(\{\!\{\_{32}, \_{32}, \_{64}\}\!\}) \ltimes [0:2] : (2)_2) \end{array} \right\}$$

△

We can now define the big-step semantics of memory patterns with the relation $\blacktriangleright$ in Fig. 3.29. We write $\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$ if the memory pattern $\widehat{p}$ matches the memory value $\widehat{v}$ considered in the store ς, and $\varsigma \vdash \widehat{p} \not\blacktriangleright \widehat{v}$ otherwise. Similarly to the high-level pattern matching judgment $\triangleright$, it proceeds by induction on $\widehat{p}$ and $\widehat{v}$, with wildcards accepting any (appropriately sized) memory value. The only subtlety is the MFission rule which, similarly to the TFission typing rule, allows recognizing mangled primitive values as integer values. Such cases arise from memory types with particularly mangled primitives such as the RISC-V layout.

$$\textsc{MWildcard}\ \frac{|\widehat{v}| \leqslant \ell}{\varsigma \vdash \__\ell \blacktriangleright \widehat{v}} \qquad \textsc{MConstant}\ \varsigma \vdash (c)_\ell \blacktriangleright (c)_\ell \qquad \textsc{MPointer}\ \frac{a \notin \text{dom}(\varsigma) \qquad \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}}{\varsigma \cup \{a \mapsto \widehat{v}\} \vdash \&_\ell(\widehat{p}) \blacktriangleright \&_\ell(a)}$$

$$\textsc{MFission}\ \frac{o_0 = 0 \qquad o_{n-1} + \ell_{n-1} = \ell \qquad o_i = o_{i-1} + \ell_{i-1} \qquad c_i = c \wedge 0\ldots0\overbrace{1\ldots1}^{\ell_i \text{ bits}}\overbrace{0\ldots0}^{o_i \text{ bits}}}{\varsigma \vdash (c)_\ell \blacktriangleright \__\ell \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \widehat{v}_i} \qquad \varsigma \vdash (c_i)_{\ell_i} \blacktriangleright \widehat{v}_i$$

$$\textsc{MComposite}\ \frac{\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \qquad \varsigma \vdash \widehat{p}_i \blacktriangleright \widehat{v}_i}{\varsigma \vdash \widehat{p} \bowtie_{0 \leqslant i < n} r_i : \widehat{p}_i \blacktriangleright \widehat{v} \bowtie_{0 \leqslant i < n} r_i : \widehat{v}_i} \qquad \textsc{MStruct}\ \frac{\varsigma \vdash \widehat{p}_i \blacktriangleright \widehat{v}_i}{\varsigma \vdash \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\} \blacktriangleright \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\}}$$

Figure 3.29: Memory-level pattern matching judgment.

Example 3.24 (Memory-level pattern matching for lists)**.** Recall the following expression from our running example:

$$e = \mathsf{let}\ x : \tau_{\text{list}}\ \mathsf{as}\ \widehat{\tau}_p = \text{Cons}(\langle 42, \text{Nil}\rangle)\ \mathsf{in}\ \mathsf{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to\ (0 : I_{32}\ \text{as}\ I_{32}) \\ \text{Cons}(\_) & \to\ (x.\text{Cons}.0 : I_{32}\ \text{as}\ I_{32}) \end{array} \right\}$$

Its reduction sequence, shown in Example 3.22, contains the following EMatch step:

$$\left\{x : \tau_{\text{list}}\ \text{as}\ \widehat{\tau}_p \mapsto \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\right\}, \mathsf{match}(x) \left\{ \begin{array}{ll} \text{Nil} & \to\ (0 : I_{32}\ \text{as}\ I_{32}) \\ \text{Cons}(\_) & \to\ (x.\text{Cons}.0 : I_{32}\ \text{as}\ I_{32}) \end{array} \right\}$$

$$\hookrightarrow_h \left\{x : \tau_{\text{list}}\ \text{as}\ \widehat{\tau}_p \mapsto \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}\right\}, (x.\text{Cons}.0 : I_{32}\ \text{as}\ I_{32})$$

To apply the EMatch rule here, we first compute the memory patterns obtained with **pat2mem** in Example 3.23. We then take the first pattern matching branch for which a memory pattern matches the memory value bound to x. Here, we take the second (Cons(_)) branch. Indeed, the only memory pattern associated with Nil does not match the memory value under scrutiny, while the one associated with the first subpattern of Cons(_) (Cons(⟨_, Nil⟩)) does:

$$\_{64} \ltimes [0:2] : (0)_2 \not\blacktriangleright \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

$$\_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : \_{32} \blacktriangleright \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32}$$

△

3.3.2.2 From pivot expressions to memory values

Let us now focus on $\hookrightarrow_m$ rules, which define the actual memory representation by actually lowering pivot expressions down to memory values. The difficulty of this lowering is that we must purposefully break type preservation during evaluation, as we are in the middle of building the memory representation of a given high-level value. This yields complex rules with intermediate steps where the local structure seems broken, but makes sense in the context of the global memory representation. In particular, for pivot expressions $(u : \tau\ \text{as}\ \widehat{\tau})$, we do *not* necessarily have agreement between τ and $\widehat{\tau}$. To keep track of such global invariants and aid the proofs later on, we will add additional artifacts in purple, which do not affect the outcome of evaluation. The first proof artifact is an additional environment σ. It is only used in accessor-related rules, which we will detail later.

The main artifacts appear in pivot expressions: in $\hookrightarrow$ rules, they are of the form $(u : \tau\ \text{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$, rather than $(u : \tau\ \text{as}\ \widehat{\tau})$. The fourth element $\widehat{\tau}_\star.\widehat{\pi}$ keeps track of the *latest well-typed state in the current evaluation sequence*, starting with $\widehat{\tau}.\varepsilon$ which is well-typed at the beginning of evaluation. For instance, if $\widehat{\tau}$ is a pointer type $\&_\ell(\widehat{\tau}')$, we progress to a pivot which focuses on the pointee while keeping the same high-level valuexpression and type, resulting in an ill-typed state. We keep track of this step by appending a pointer dereference to $\widehat{\pi}$, yielding the following pivot expression: $(u : \tau\ \text{as}\ \widehat{\tau}' \equiv \widehat{\tau}_\star.\widehat{\pi}.*)$. The appearance of ill-typed states in inevitable, given that we define our semantics as a sequence of tiny steps, which by design model unfinished memory values. We reset $\widehat{\tau}_\star.\widehat{\pi}$ at fragments, which correspond to explicit synchronization points between high-level and memory types. More generally, we maintain the following invariant: in every pivot expression $(u : \tau\ \text{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$, we have $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_\star) = \widehat{\tau}$ and $\mathbf{agree}_\Delta(\tau, \widehat{\tau}_\star)$.

We now look at each rule in Fig. 3.30 one by one, starting with the two in Fig. 3.30a. Memory contexts, denoted C indicate the position of a hole □ within a memory expression. They are only used to state the ESubStep rule, which covers evaluation steps on nested sub-expressions, both within memory structures and as let-bound expressions. Note that contexts do not mandate an evaluation order. The rule EAddress finalizes the construction of memory values by lifting inlined pointer contents outside of the memory valuexpression and into the store, using a fresh address.

(a) Memory contexts and non-pivot rules

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

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

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

(b) Memory structures

ETypeVar
$$\frac{(t \mapsto \widehat{\tau}) \in \Delta \qquad \Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma', \sigma', \widehat{\sigma}', \varsigma', \widehat{e}}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ t \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma', \sigma', \widehat{\sigma}', \varsigma', \widehat{e}}$$

EConstant
$$\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ (c)_\ell \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, (c)_\ell$$

EWord
$$\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \__\ell \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, \__\ell$$

EPointer
$$\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \&_\ell\,(\widehat{\tau}) \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, \&_\ell\,((u : \tau\ \mathsf{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}.*))$$

EComposite
$$\frac{\widehat{\pi}' = \widehat{\pi}.\neg r_0 \ldots \neg r_{n-1} \qquad \widehat{\pi}_i = \widehat{\pi}.r_i}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}') \bowtie_{0 \leqslant i < n} r_i : (u : \tau\ \mathsf{as}\ \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}_i)}$$

EStruct
$$\frac{\widehat{u}_i = (u : \tau\ \mathsf{as}\ \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}.i)}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, \{\!\{\widehat{u}_0, \ldots, \widehat{u}_{n-1}\}\!\}}$$

(c) Synchronization between high-level and memory types

ESplit
$$\frac{\widehat{\tau} = \mathrm{split}\,(\ldots) \qquad \exists (p, \widehat{\tau}') \in \widehat{\tau}/\_.\Delta, \Gamma \vdash u : \tau/p \qquad \widehat{\tau}'_\star = \widehat{\tau}_\star\,[.\widehat{\pi} \leftarrow \widehat{\tau}']}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau/p\ \mathsf{as}\ \widehat{\tau}' \equiv \widehat{\tau}'_\star.\widehat{\pi})}$$

EFragment
$$u' = \begin{cases} \textbf{focus}\,(\pi, u) & \text{if defined} \\ \text{any inhabitant of the type } \textbf{focus}\,(\pi, \tau) & \text{otherwise} \end{cases}$$
$$\frac{}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau\ \mathsf{as}\ (\pi\ \mathsf{as}\ \widehat{\tau}) \equiv \widehat{\tau}_\star.\widehat{\pi}) \hookrightarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u' : \textbf{focus}\,(\pi, \tau)\ \mathsf{as}\ \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)}$$

Figure 3.30: Memory-level expression evaluation, computation of memory values. Environments appearing in gray in a rule are irrelevant to its application. Purple elements are only useful for stating and proving type soundness.

(d) Subterm extraction

EVarAccess

$$\frac{(x : \tau_x \text{ as } \widehat{\tau}) \in \Gamma}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (x.\varepsilon : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \rightsquigarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, \widehat{\sigma}(x)}$$

EVarFocus

$$\frac{\begin{array}{c}(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma \qquad (p_b, \widehat{\tau}_b) \in \widehat{\tau}_x \big/ \_ \qquad \Delta, \Gamma, \varsigma \vdash \widehat{\sigma}(x) : \widehat{\tau}_b \\ (\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathbf{shatter}(\widehat{\tau}_b) \qquad \tau_f = \mathbf{focus}\,(\pi_f, \tau_x) \qquad \widehat{v}_f = \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) \qquad x_f \text{ fresh symbol}\end{array}}{\begin{array}{l}\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (x.(\pi_f.\pi) : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \\ \quad \rightsquigarrow_m \Gamma \cup \{(x_f : \tau_f \text{ as } \widehat{\tau}_f)\}, \sigma \cup \{x_f \mapsto x.\pi_f\}, \widehat{\sigma} \cup \{x_f \mapsto \widehat{v}_f\}, \varsigma, (x_f.\pi : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})\end{array}}$$

(e) Primitive types

EAtom

$$\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (c : \tau \text{ as } I_\ell \equiv \widehat{\tau}_\star.\widehat{\pi}) \rightsquigarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, (c)_\ell$$

EFission

$$\frac{r_i = [o_i : \ell_i] \qquad o_0 = 0 \qquad o_{n-1} + \ell_{n-1} = \ell \qquad o_{i-1} + \ell_{i-1} = o_i}{\Delta, \Sigma \vdash \Gamma, \sigma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } I_\ell \equiv \widehat{\tau}_\star.\widehat{\pi}) \rightsquigarrow_m \Gamma, \sigma, \widehat{\sigma}, \varsigma, \__\ell \bowtie_{0 \leqslant i < n} r_i : (\mathbf{focus}\,(.r_i, u) : \mathbf{focus}\,(.r_i, \tau) \text{ as } I_{\ell_i} \equiv \widehat{\tau}_\star.\widehat{\pi}.r_i)}$$

Figure 3.30: (continued). Memory-level expression evaluation, computation of memory values. Environments appearing in gray in a rule are irrelevant to its application. Purple elements are only useful for stating and proving type soundness.

The goal of all remaining rules is to reduce a given pivot expression $(u : \tau \text{ as } \widehat{\tau})$ to a memory value representing u using the layout $\widehat{\tau}$. This task relies on four kinds of rules.

- The most straightforward rules process *memory structures* such as structs, composite words or pointers. These rules, defined in Fig. 3.30b, inspect the shape of $\widehat{\tau}$ and distribute u (and τ) over its components. The result is a memory valuexpression in which the root memory construct has been lifted from the type to the value itself. We then proceed by induction on each component of the memory structure.
- As seen before, memory types do not only consist of memory structures. Some key constructs act as *synchronization points* between an ADT and its memory layout, namely fragments, splits. We handle such memory types with ESplit and EFragment, defined in Fig. 3.30c.
  - For splits, we proceed by specializing the memory type, then selecting the branch whose provenance matches the considered valuexpression. There always exists exactly one such branch if the valuexpression is well-typed.
  - The semantics of a fragment type $(\pi \text{ as } \widehat{\tau})$ is "represent the subterm at position π within the high-level value according to the memory layout $\widehat{\tau}$". Accordingly, EFragment creates a new pivot expression by focusing the high-level valuexpression and type on π and using the specified memory type $\widehat{\tau}$. While $\mathbf{focus}\,(\pi, \tau)$ is always defined owing to the fragment coherence agreement criterion, $\mathbf{focus}\,(\pi, u)$ may not be. Indeed, it is technically possible for fragments to refer to any high-level constructor τ, even outside of a split branch that specifically restricts possible values to this constructor. For instance, the memory type $\left\{\!\!\left\{\text{split}\,(\varepsilon) \left\{ \begin{array}{rcl} 0 & \text{from} & A \Rightarrow (0)_8 \\ 1 & \text{from} & B \Rightarrow (1)_8 \end{array} \right\}, (.A \text{ as } I_{32}), (.B \text{ as } I_{32}) \right\}\!\!\right\}$ is well-kinded and agrees with the high-level type $A(I_{32}) \mid B(I_{32})$, yet the subterm at position $.B$ is undefined for values of the form $A(v)$ and vice-versa. In this case, we simply use any inhabitant of τ as the new valuexpression – given our ADT grammar, it is easy to find such a value for any high-level type.

- Unlike high-level values, for which we can build a memory value by induction on the memory type using the previous two kinds of rules, evaluating an accessor $x.\pi$ according to the memory type $\widehat{\tau}$ involves a combination of two tasks:
  - extracting the parts relevant to π from the memory representation of the value bound to x;
  - rearranging these parts to fit the desired layout $\widehat{\tau}$.

  The first task is performed by two *bound value extraction* rules, defined in Fig. 3.30d. In the simplest case (handled by EVarAccess), π is empty and the existing memory representation of x follows $\widehat{\tau}$. In this case, the desired memory value has already been computed and stored as $\widehat{\sigma}(x)$. Otherwise, we use the EVarFocus rule to inspect the layout of x, denoted $\widehat{\tau}_x$, in order to find the parts of $\widehat{\sigma}(x)$ that are relevant to $x.\pi$. More formally, we are looking for a fragment (or a primitive type) which represents the subterm at position π (or a prefix of π) within x. To this end, we use the **shatter** operation defined in Fig. 3.13 to gather all fragments and primitive types in $\widehat{\tau}_x$, then filter these to keep prefixes of π. Before using **shatter**, we must remove splits from $\widehat{\tau}_x$ by specializing it for the wildcard pattern _, yielding a set of branches consisting of a more precise pattern and a split-free memory type. Assuming that $\widehat{\sigma}(x)$ is of type $\widehat{\tau}_x$, there exists at least one branch $(p_b, \widehat{\tau}_b)$ such that $\widehat{\sigma}(x)$ is of type $\widehat{\tau}_b$ (and the original high-level value represented by $\widehat{\sigma}(x)$ matches p_b). Conversely, no memory value can belong to more than one specialized type, owing to the distinguishability agreement criterion. Therefore, there exists a unique branch that matches $\widehat{\sigma}(x)$; we select this branch and **shatter** its memory type to search for a suitable fragment (or primitive type). If such a fragment exists, the EVarFocus rule applies: we create an intermediate value binding this part of $\widehat{\sigma}(x)$ and attempt to extract the desired piece of data from this new memory value. We keep also track of this intermediate value in the high-level binding environment σ, which will only be used for proofs.

  The second task is performed when neither of these two rules apply: we have to break down $\widehat{\tau}$ using other rules. Termination relies on the coverage agreement criterion: $\widehat{\tau}_x$ must represent the subterm $.\pi$ in some form, although it may break it down into smaller pieces represented at different locations. Therefore, this process will eventually lead to an accessor for which the EVarAccess rule applies. The same problem is encountered during compilation of valuexpressions, and we solve it in a very similar way in our compilation approach, as we will see in Chapter 5.

- Primitive (integer) types I_ℓ are handled by two different rules, defined in Fig. 3.30e, depending on the high-level valuexpression. EAtom handles primitive constants by encoding them on ℓ bits. While primitive types are usually "atomic", in that there is usually no need to decompose them further, this is not always the case. For instance, consider the pivot expression $(x.\varepsilon : I_{64} \text{ as } I_{64})$ with the typing environment $\{(x : I_{64} \text{ as } \{\!\{(x.[0:32] \text{ as } I_{32}), (x.[32:32] \text{ as } I_{32})\}\!\})\}$. Here, x refers to a 64-bit integer whose two 32-bit halves are represented as separate fields in a struct. Since $x.\varepsilon$ is not a constant, EAtom does not apply, and since no toplevel primitive type or ε-fragment appears in the memory type used for x, neither do EVarAccess and EVarFocus. Instead, we must extract both 32-bit fragments and recombine them into a single 64-bit integer. EFission lets us break down the primitive type I_{64} into a combination of two I_{32} parts, yielding the following expression: $\__{64} \ltimes [0:32] : (x.[0:32] : I_{32} \text{ as } I_{32}) \ltimes [32:32] : (x.[32:32] : I_{32} \text{ as } I_{32})$. Both of the two new pivots that appear in it can be reduced with EVarFocus. More generally, EFission allows to partition a primitive type into any number of consecutive bit ranges, so as to rebuild an integer value piece by piece when necessary. Its counterpart is the memory typing rule TFission, which interprets the resulting composite words as proper integers.

Example 3.25 (Legal, but temporarily ill-typed expression)**.** Recall the memory-level evaluation sequence from Example 3.22. We focus on the first let-bound pivot expression, and extend it with a fourth field initialized with $\widehat{\tau}_p.\varepsilon$ since this pivot is well-typed:

$$(\text{Cons}(\langle 42, \text{Nil}\rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p \equiv \widehat{\tau}_p.\varepsilon)$$

Let

$$\widehat{\tau}_b = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons}.0 \text{ as } I_{32})$$

The evaluation sequence for this pivot is:

$$
\begin{aligned}
&\quad (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \tau_{\mathrm{list}} \text{ as } \widehat{\tau}_p \equiv \widehat{\tau}_p.\varepsilon) \\
&\hookrightarrow_m (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } \widehat{\tau}_b \equiv \widehat{\tau}_b.\varepsilon) && \text{(ESplit)} \\
&\hookrightarrow_m (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } \_{64} \equiv \widehat{\tau}_b.\neg[0:2].\neg[2:32]) \\
&\quad \ltimes [0:2] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (1)_2 \equiv \widehat{\tau}_b.[0:2]) \\
&\quad \ltimes [2:32] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (.\mathrm{Cons}.0 \text{ as } I_{32}) \equiv \widehat{\tau}_b.[2:32]) \\
& && \text{(EComposite)} \\
&\hookrightarrow_{m\_64} \ltimes [0:2] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (1)_2 \equiv \widehat{\tau}_b.[0:2]) \\
&\quad \ltimes [2:32] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (.\mathrm{Cons}.0 \text{ as } I_{32}) \equiv \widehat{\tau}_b.[2:32]) \\
& && \text{(EWord)} \\
&\hookrightarrow_{m\_64} \ltimes [0:2] : (1)_2 \\
&\quad \ltimes [2:32] : (\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (.\mathrm{Cons}.0 \text{ as } I_{32}) \equiv \widehat{\tau}_b.[2:32]) \\
& && \text{(EConstant)} \\
&\hookrightarrow_{m\_64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42 : I_{32} \text{ as } I_{32} \equiv I_{32}.\varepsilon) && \text{(EFragment)} \\
&\hookrightarrow_{m\_64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (42)_{32} && \text{(EAtom)}
\end{aligned}
$$

Notice how several pivot expressions that appear within this sequence are not well-typed when we only consider their first three components. For instance, the EComposite step introduces the two following pivots:

$$
\begin{gathered}
(\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } \_{64} \equiv \widehat{\tau}_b.\neg[0:2].\neg[2:32]) \\
(\mathrm{Cons}(\langle 42, \mathrm{Nil}\rangle) : \mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle) \text{ as } (1)_2 \equiv \widehat{\tau}_b.[0:2])
\end{gathered}
$$

Neither $\_{64}$ nor $(1)_2$ agree with the high-level type $\mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle)$ that appears in these pivots; therefore, they are not well-typed according to the original high-level typing judgment. However, their fourth component keeps track of the original memory type $\widehat{\tau}_b$, of which $\_{64}$ and $(1)_2$ are subterms, which does agree with $\mathrm{Cons}(\langle I_{32}, \mathrm{Nil}\rangle)$. △

3.4 Memotheory

We now state and prove properties of our semantics to ensure that the memory-level and high-level behaviors of ribbit programs are coherent. The most important result of this chapter is a proof of *branching bisimulation* (Glabbeek and Weijland 1996; De Nicola and Vaandrager 1995) between $\hookrightarrow$ and $\rightsquigarrow$ (Theorem 3.3). This concept is illustrated in Fig. 3.31. For this bisimulation, each step of the high-level reduction sequence using the $\hookrightarrow$ rule labelled l_i has a counterpart in the memory-level reduction sequence which uses the corresponding $\rightsquigarrow$ rule labelled h, l_i (the h indicates that this is indeed a "high-level" step). Between these synchronized steps, the memory-level reduction sequence may go through $\hookrightarrow_m$ steps, which correspond to と-transitions [4], i.e., silent transitions which have no equivalent in the high-level reduction sequence but preserve the bisimulation relation $\mathcal{R}$ with the current high-level expression. In essence, we show that the traditional high-level semantics is replayed exactly during low-level evaluation, with some additional steps interspersed to build memory values.

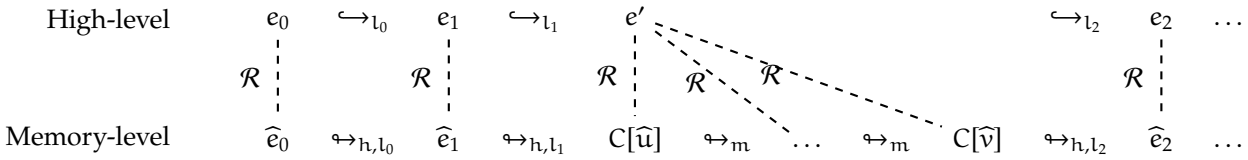


Figure 3.31: Diagram showing branching bisimulation between high-level and memory reduction sequences of the same source expression.

[4] Usually denoted τ-transitions in most other contexts. However, as the τ symbol is rather overloaded in this thesis, we use the Japanese hiragana character と (pronounced [to]) to denote silent transitions.

We first extend evaluation and typing, along with some notations, to help with the proofs (Section 3.4.1). In Section 3.4.2, we define a *simulation relation* between evaluation states by combining the typing judgements with a new notion of a memory value accurately *representing* a high-level value according to a given memory layout. Finally, in Section 3.4.4, we leverage the previously defined tools to show our results:

- correctness of pattern matching (Theorem 3.2);
- progress and preservation of memory evaluation (Lemmas 3.13 and 3.14);
- the branching bisimulation between $\hookrightarrow$ and $\rightarrowtail$ (Lemma 3.15 and Theorem 3.3).

3.4.1 Expanded Judgements and notations

3.4.1.1 Labelled Transitions

We aim to show preservation of evaluation between high-level evaluation steps. We must thus equip our transitions with *labels* that will be preserved by bisimulation. In the rest of this section, we label high-level evaluation steps with the name of their rules: for each $\mathfrak{l} \in \{\text{EFunCall}, \text{ELetBind}, \text{EMatch}\}$, we write $\hookrightarrow_{\mathfrak{l}}$ for high-level evaluation steps which use the rule HL$\mathfrak{l}$ and $\rightarrowtail_{\mathfrak{hl}}$ for memory-level evaluation steps with use the rule $\mathfrak{l}$. On the other hand, $\rightarrowtail_{\mathfrak{m}}$ steps do not carry any additional label.

3.4.1.2 Typing judgment extension

As seen in Section 3.3.2, some intermediate stages reached by memory evaluation are not well-typed w.r.t. the current memory typing judgment, even though they always eventually reach a well-typed state. In order to reason on these ill-typed expressions, we must relax the well-typed criterion while still constraining expressions enough to ensure correctness. To this end, we annotated each pivot $(u : \tau \text{ as } \widehat{\tau})$ with a parent memory type $\widehat{\tau}_\star$ and a memory path $\widehat{\pi}$ such that $\widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_\star) = \widehat{\tau}$ and which represents "the latest well-typed state". Formally, we have $\vdash (u : \tau \text{ as } \widehat{\tau}_\star) : \widehat{\tau}_\star$, but not necessarily $\vdash (u : \tau \text{ as } \widehat{\tau}) : \widehat{\tau}$. We use $\widehat{\tau}_\star$ for typing and $\widehat{\tau}$ for evaluation. We broaden the TPivot rule of the typing judgment for expressions defined in Fig. 3.23:

$$\textsc{TPivot}\ \frac{\dfrac{\textbf{agree}_\Delta(\tau, \widehat{\tau}_\star) \qquad \Delta, \Gamma \vdash u : \tau}{\Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau}_\star) : (\tau \text{ as } \widehat{\tau}_\star)} \qquad \widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_\star) = \widehat{\tau}}{\Delta, \Gamma, \varsigma \vdash (u : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) : \tau \text{ as } \widehat{\tau}}$$

3.4.1.3 Notational relief

We define the following notational shorthands for well-formed and well-typed environments, expressions and pivots. Each combines an existing judgement, such as typing, with validity of all its premises.

$$\vDash \Delta, \Gamma \vdash \sigma \iff \vDash \Delta \wedge \Delta \vDash \Gamma \wedge \Delta, \Gamma \vdash \sigma \qquad \text{(Validity of evaluation contexts)}$$

$$\vDash \Delta, \Gamma, \varsigma \vdash \widehat{\sigma} \iff \vDash \Delta \wedge \Delta \vDash \Gamma \wedge \Delta, \Gamma, \varsigma \vdash \widehat{\sigma} \qquad \text{(Validity of memory evaluation contexts)}$$

$$\vDash \Delta, \Gamma \vdash e : \tau \iff \vDash \Delta \wedge \Delta \vDash \Gamma \wedge \Delta, \Gamma \vdash e : \tau \qquad \text{(Validity and typing)}$$

$$\vDash \Delta, \Gamma, \varsigma \vdash \widehat{e} : \widehat{\tau} \iff \vDash \Delta \wedge \Delta \vDash \Gamma \wedge \Delta \vDash \widehat{\tau} \wedge \Delta, \Gamma, \varsigma \vdash \widehat{e} : \widehat{\tau} \qquad \text{(Validity and memory typing)}$$

$$\Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau}) \iff \Delta \vDash \widehat{\tau} \wedge \textbf{agree}_\Delta(\tau, \widehat{\tau}) \wedge \Delta, \Gamma \vdash u : \tau \qquad \text{(Validity of pivot expression)}$$

$$\Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \iff \Delta, \Gamma \vdash (u : \tau \text{ as } \widehat{\tau}_\star) \wedge \Delta \vDash \widehat{\tau} \wedge \widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_\star) = \widehat{\tau}$$
(Validity of annotated pivot expr.)

3.4.2 Simulation and representation relations

We now define our simulation relation denoted $\mathcal{R}$ in Fig. 3.32, which underpins our branching bisimulation theorem. We first introduce a relation between high-level expressions e and memory expressions $\widehat{e}$ defined in Fig. 3.32a and denoted $\Delta, \Gamma, \sigma, \varsigma \vdash e \mathcal{R} \widehat{e}$. We then extend it to high-level evaluation states $S = (\Gamma_h, \sigma_h, e)$ and memory evaluation states $\widehat{S} = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{e})$ in Fig. 3.32b.

$$\Delta, \Gamma, \sigma, \varsigma \vdash e \mathcal{R} e \qquad \frac{\Delta, \Gamma, \sigma, \varsigma \vdash e \mathcal{R} \widehat{e}}{\Delta, \Gamma, \sigma, \varsigma \vdash \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = e\ \mathsf{in}\ e_0 \mathcal{R} \mathsf{let}\ x : \tau\ \mathsf{as}\ \widehat{\tau} = \widehat{e}\ \mathsf{in}\ e_0}$$

$$\frac{\Delta, \varsigma \vdash \widehat{u}[\![\sigma]\!]\ \mathrm{reprs}\ (u[\![\sigma]\!] : \tau\ \mathrm{as}\ \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)}{\Delta, \Gamma, \sigma, \varsigma \vdash (u : \tau\ \mathrm{as}\ \widehat{\tau})\ \mathcal{R}\ \widehat{u}}$$

(a) Relation between expressions

$$\frac{\begin{array}{c}\vDash \Delta, \Gamma_h \vdash \sigma_h \qquad \Delta, \Gamma_h \vdash \Sigma \qquad \Gamma_h \subseteq \Gamma_m \qquad \vDash \Delta, \Gamma_m, \varsigma \vdash \widehat{\sigma}\ \mathrm{reprs}\ (\sigma_h \sqcup \sigma_m) \\ \mathbf{agree}_\Delta(\tau, \widehat{\tau}) \qquad \vDash \Delta, \Gamma_h \vdash e : \tau\ \mathrm{as}\ \widehat{\tau} \qquad \vDash \Delta, \Gamma_m, \varsigma \vdash \widehat{e} : \widehat{\tau} \qquad \Delta, \Gamma_m, \sigma_h \sqcup \sigma_m, \varsigma \vdash e \mathcal{R} \widehat{e}\end{array}}{\Delta, \Sigma, \tau, \widehat{\tau} \vdash (\Gamma_h, \sigma_h, e)\ \mathcal{R}\ (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{e})}$$

(b) Relation between full states

Figure 3.32: Simulation relation between high-level and memory evaluation states.

The first cases (Fig. 3.32a) apply to memory expressions that contain high-level constructs. They ensure that these constructs are exactly the same as those found in the high-level expression e. The case of Fig. 3.32bis more complex: it describes the memory-level stage of evaluation, when e has reached a normal form for $\hookrightarrow$ (that is, a pivot expression $(u : \tau\ \mathrm{as}\ \widehat{\tau})$) and memory-level evaluation (i.e., $\twoheadrightarrow_m$ steps) of $\widehat{e}$ is underway. It requires the notion of a memory value *representing* a given high-level value according to a memory layout. This is captured by the *reprs* relation defined in Fig. 3.33. It is a syntactical characterization of memory valuexpressions which are reachable via $\twoheadrightarrow_m$ steps from a given pivot $(u : \tau\ \mathrm{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$. We write $\Delta, \varsigma \vdash \widehat{u}\ \mathrm{reprs}\ (v : \tau\ \mathrm{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$ when the memory valuexpression $\widehat{u}$ represents the high-level value v of type τ according to the memory type $\widehat{\tau}$, which is at position $\widehat{\pi}$ within the latest well-typed layout $\widehat{\tau}_\star$. The *reprs* relation is defined for *normalized* valuexpressions: we assume that v, as well as the first component of every pivot appearing within $\widehat{u}$, are values (as opposed to valuexpressions containing variables). Given a high-level binding environment σ, $u[\![\sigma]\!]$ denotes the substitution of every variable in u with its bound value in σ, and $\widehat{u}[\![\sigma]\!]$ denotes $\widehat{u}$ in which the valuexpression of every pivot has been normalized in this way. Thanks to this relation, we capture all intermediate stages of the $\twoheadrightarrow_m$ reduction sequence from $(v : \tau\ \mathrm{as}\ \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$ to a memory value. The base case is RIdentity; all other rules correspond to a $\twoheadrightarrow_m$ evaluation rule. Most rules are straightforward syntactical translations of their evaluation counterparts, with some simplifications (as they simply relate existing expressions and do not need to construct a new state).

RIdentity
$$\Delta, \varsigma \vdash (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$$

RTypeVar
$$\frac{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \Delta(t) \equiv \widehat{\tau}_\star.\widehat{\pi})}{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } t \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RAddress
$$\frac{a \notin \varsigma \qquad \Delta, \varsigma \vdash \&_\ell\,(\widehat{v}) \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})}{\Delta, \varsigma \cup \{a \mapsto \widehat{v}\} \vdash \&_\ell\,(a) \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RAtom
$$\Delta, \varsigma \vdash (c)_\ell \text{ reprs } (c : \tau \text{ as } I_\ell \equiv \widehat{\tau}_\star.\widehat{\pi})$$

RFission
$$\frac{o_0 = 0 \qquad o_{n-1} + \ell_{n-1} = \ell \qquad o_i = o_{i-1} + \ell_{i-1} \qquad \Delta, \varsigma \vdash \widehat{u}_i \text{ reprs } (\textbf{focus}\,(.r_i, v) : \textbf{focus}\,(r_i, \tau) \text{ as } I_{\ell_i} \equiv \widehat{\tau}_\star.\widehat{\pi}.r_i)}{\Delta, \varsigma \vdash \__\ell \bowtie_{0 \leqslant i < n} r_i : \widehat{u}_i \text{ reprs } (v : \tau \text{ as } I_\ell \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RFragment
$$\frac{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (\textbf{focus}\,(\pi, v) : \textbf{focus}\,(\pi, \tau) \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)}{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } (\pi \text{ as } \widehat{\tau}) \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RSplit
$$\frac{\widehat{\tau} = \text{split}\,(\dots) \qquad \exists (p, \widehat{\tau}') \in \widehat{\tau}/\_.\Delta \vdash v : \tau/p \qquad \Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau/p \text{ as } \widehat{\tau}' \equiv \widehat{\tau}_\star\,[.\widehat{\pi} \leftarrow \widehat{\tau}'].\widehat{\pi})}{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RWord
$$\Delta, \varsigma \vdash \__\ell \text{ reprs } (v : \tau \text{ as } \__\ell \equiv \widehat{\tau}_\star.\widehat{\pi})$$

RConstant
$$\Delta, \varsigma \vdash (c)_\ell \text{ reprs } (v : \tau \text{ as } (c)_\ell \equiv \widehat{\tau}_\star.\widehat{\pi})$$

RPointer
$$\frac{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}.*)}{\Delta, \varsigma \vdash \&_\ell\,(\widehat{u}) \text{ reprs } (v : \tau \text{ as } \&_\ell\,(\widehat{\tau}) \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RComposite
$$\frac{\Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}.\neg r_0.\dots.\neg r_{n-1}) \qquad \Delta, \varsigma \vdash \widehat{u}_i \text{ reprs } (v : \tau \text{ as } \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}.r_i)}{\Delta, \varsigma \vdash \widehat{u} \bowtie_{0 \leqslant i < n} r_i : \widehat{u}_i \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

RStruct
$$\frac{\Delta, \varsigma \vdash \widehat{u}_i \text{ reprs } (v : \tau \text{ as } \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}.i)}{\Delta, \varsigma \vdash \{\!\{\widehat{u}_0, \dots, \widehat{u}_{n-1}\}\!\} \text{ reprs } (v : \tau \text{ as } \{\!\{\widehat{\tau}_0, \dots, \widehat{\tau}_{n-1}\}\!\} \equiv \widehat{\tau}_\star.\widehat{\pi})}$$

Figure 3.33: Representation relation between *normalized* (i.e., variable-free) pivot expressions and memory valuexpressions.

Finally, we extend this relation to environments. Given high-level and memory-level binding environments σ and $\widehat{\sigma}$, we write $\Delta, \Gamma, \varsigma \vdash \widehat{\sigma} \text{ reprs } \sigma$ if we have $\text{dom}\,(\widehat{\sigma}) = \text{dom}\,(\sigma) = \text{dom}\,(\Gamma)$ and for each $(x : \tau \text{ as } \widehat{\tau}) \in \Gamma$, $\Delta, \varsigma \vdash \widehat{\sigma}(x)[\![\sigma]\!] \text{ reprs } (\sigma(x)[\![\sigma]\!] : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)$.

As before, we also define the following syntactical shorthands for the representation relation with a well-typed pivot expression in well-formed and well-typed environments:

$$\vDash \Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \iff \Delta \vdash (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \wedge \Delta, \varsigma \vdash \widehat{u} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$$
(Validity and reprs)

$$\vDash \Delta, \Gamma, \varsigma \vdash \widehat{\sigma} \text{ reprs } \sigma \iff \text{dom}\,(\sigma) = \text{dom}\,(\widehat{\sigma}) = \text{dom}\,(\Gamma) \wedge \forall (x : \tau \text{ as } \widehat{\tau}) \in \Gamma, \vDash \Delta, \varsigma \vdash \widehat{\sigma}(x)[\![\sigma]\!] \text{ reprs } (\sigma(x)[\![\sigma]\!] : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)$$
(Validity and Environment reprs)

3.4.3 Results on high-level and memory-level pattern matching

As a prerequisite for our main results on $\hookrightarrow$ and $\twoheadrightarrow$, we first establish an equivalence between high-level and memory-level pattern matching through **pat2mem** and our *reprs* relation. The main result of this section is Theorem 3.2. To prove it, we will use alternative characterizations of both high-level (Lemma 3.5) and memory-level (Lemma 3.6) pattern matching judgments. Along with agreement criteria between high-level and memory types, these will allow us to show that every part of a given high-level pattern corresponds to specific parts of its memory counterparts obtained with **pat2mem**.

In this section, several results will be proven by induction on a pair $(p, \widehat{\tau})$ consisting of a high-level pattern and a memory type. To ensure this induction is well-founded, we assume that all fragments of the form $(\varepsilon \text{ as } \widehat{\tau}_f)$ appearing in memory types have been replaced with $\widehat{\tau}_f$. This unrolling of epsilon-fragments always terminates for the memory types we consider (indeed, memory types containing cycles of such epsilon-fragments have limited practical interest: such types do not have a computable size or shape, nor any finite inhabitant).

We also relax the typing judgment for memory patterns to allow for (adequately sized) wildcards. In this section, preconditions and conclusions of the form $\Delta \vdash \widehat{p} : \widehat{\tau}$ may use the following rule in addition to existing memory typing rule:

$$\frac{|\widehat{\tau}| \leqslant \ell}{\Delta, \Gamma, \varsigma \vdash \__\ell : \widehat{\tau}} \textsc{MemTWildcard}$$

Lemma 3.5 (Characterization of high-level pattern matching)**.** *Let Δ, τ, p and v such that*

$$\vDash \Delta \vdash p : \tau \qquad\qquad \vDash \Delta \vdash v : \tau$$

We have $p \rhd v$ if and only if both of the following conditions hold:

Each individual bit of every primitive matches:

$$\forall \pi \in \mathit{Paths}, (\mathbf{focus}_\Delta(\pi, \tau) = I_1 \wedge \mathbf{focus}(\pi, p) = c \wedge \mathbf{focus}(\pi, v) = c') \Rightarrow c = c'$$

Each head constructor matches:

$$\forall \pi \in \mathit{Paths}, \left(\mathbf{focus}_\Delta(\pi, \tau) = \Big|_{0 \leqslant i < n} K_i(\tau_i) \wedge \mathbf{focus}(\pi, p) = K(p') \wedge \mathbf{focus}(\pi, v) = K'(v')\right) \Rightarrow K = K'$$

Proof. Immediate by induction. □

The following result states that a memory pattern matches a memory value of the same type if and only if both belong to the same specialized branch of this type and all of their parts corresponding to a fragment or primitive in their type (as gathered by **shatter**) match.

Lemma 3.6 (Characterization of memory-level pattern matching)**.** *Let Δ, ς, $\widehat{\tau}$, $\widehat{p}$ and $\widehat{v}$ such that*

$$\vDash \Delta \vdash \widehat{p} : \widehat{\tau} \qquad\qquad \vDash \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}$$

We have $\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$ if and only if there exists a branch $(p_b, \widehat{\tau}_b) \in \widehat{\tau}/\_$ such that

$$\Delta \vdash \widehat{p} : \widehat{\tau}_b \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}_b \qquad \forall (\widehat{\pi} \mapsto \pi \text{ as } \widehat{\tau}') \in \mathbf{shatter}_\Delta(\widehat{\tau}_b), \varsigma \vdash \widehat{\mathbf{focus}}(\widehat{\pi}, \widehat{p}) \blacktriangleright \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{v})$$

Proof. Immediate by induction on $\widehat{\tau}$. □

Through Lemma 3.5 (resp. Lemma 3.6), we have established a correspondence between high-level (resp. memory-level) pattern matching and specific locations within high-level (resp. memory) types consistent with agreement criteria. In order to use this correspondence to prove our main result (Theorem 3.2), we need to be able to "reach" these locations within values and patterns. We do so using the following Lemmas 3.7 to 3.10, which let us synchronize the exploration of memory types with that of memory patterns obtained through **pat2mem** and of memory values representing a given high-level value.

Lemma 3.7 (Matching type branches for **pat2mem**). *Let* $\Delta, \widehat{\tau}, p_b, \widehat{\tau}_b, p, p'$ *and* $\widehat{p}$ *such that*

$$\Delta \vDash \widehat{\tau} \qquad (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}) \qquad (p_b, \widehat{\tau}_b) \in \widehat{\tau}/\_ \qquad \Delta \vdash \widehat{p} : \widehat{\tau}_b$$

The pattern intersection $p_b \sqcap p'$ *is defined and we have*

$$(p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}_b)$$

Proof. Immediate by induction on $(p, \widehat{\tau})$. □

Lemma 3.8 (Memory focusing and **pat2mem** commute). *Let* $\Delta, \widehat{\tau}, p, p', \widehat{p}$ *and* $\widehat{\pi}$ *such that*

$$\Delta \vDash \widehat{\tau} \qquad (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}) \qquad \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) \text{ is defined}$$

Either p *is a wildcard pattern* $\_$ *or there exists* p'' *such that*

$$\left(p'', \widehat{\mathbf{focus}}(\widehat{\pi}, \widehat{p})\right) \in \mathbf{pat2mem}_\Delta(p, \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}))$$

Proof. Immediate by induction on $(p, \widehat{\tau})$. □

Lemma 3.9 (Matching type branches for value memory representations). *Let* $\Delta, \varsigma, \tau, \widehat{\tau}, \widehat{\tau}_\star, \widehat{\pi}_\star, p_b, \widehat{\tau}_b, v$ *and* $\widehat{v}$ *such that*

$$\vDash \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}_\star) \qquad (p_b, \widehat{\tau}_b) \in \widehat{\tau}/\_ \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}_b$$

We have

$$\Delta \vdash v : \tau/p_b \qquad p_b \rhd v \qquad \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau/p_b \text{ as } \widehat{\tau}_b \equiv \widehat{\tau}_\star[.\widehat{\pi}_\star \leftarrow \widehat{\tau}_b].\widehat{\pi}_\star)$$

Proof. Immediate by induction on $(v, \widehat{\tau})$. □

Lemma 3.10 (Memory focusing and *reprs* commute on memory values). *Let* $\Delta, \varsigma, \tau, \widehat{\tau}, \widehat{\tau}_\star, \widehat{\pi}_\star, v, \widehat{v}$ *and* $\widehat{\pi}$ *such that*

$$\vDash \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}_\star) \qquad \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) \text{ is defined}$$

We have

$$\Delta, \varsigma \vdash \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{v}) \text{ reprs } (v : \tau \text{ as } \widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}) \equiv \widehat{\tau}_\star.\widehat{\pi}_\star.\widehat{\pi})$$

Proof. Immediate by induction on $(v, \widehat{\tau})$. □

We can now state our main result on **pat2mem**.

Theorem 3.2 (**pat2mem** correctness). *Let* $\Delta, \varsigma, \tau, \widehat{\tau}, p, v$ *and* $\widehat{v}$ *such that*

$$\vDash \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon) \qquad \Delta \vdash p : \tau$$

We have

$$p \rhd v \iff \exists (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}),\ \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$$

Proof. We prove each direction of the equivalence in the following Lemmas 3.11 and 3.12. □

Lemma 3.11 (High-level matching implies memory matching). *Let* $\Delta, \varsigma, \tau, \widehat{\tau}, \widehat{\tau}_\star, \widehat{\pi}, p, v$ *and* $\widehat{v}$ *such that*

$$\vDash \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau} \qquad \Delta \vdash p : \tau \qquad p \rhd v$$

There exist p' *and* $\widehat{p}$ *such that*

$$(p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}) \qquad p' \rhd v \qquad \Delta \vdash \widehat{p} : \widehat{\tau} \qquad \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$$

Proof. Let us proceed by induction on $(p, \widehat{\tau})$.

Wildcard: $p = \_$**.** Immediate: we have

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \{(\_, \_|\widehat{\tau}|)\} \qquad \_ \rhd \nu \qquad \Delta \vdash \_|\widehat{\tau}| : \widehat{\tau} \text{ (MemTWildcard)} \qquad \varsigma \vdash \_|\widehat{\tau}| \blacktriangleright \widehat{\nu} \text{ (MWildcard)}$$

Primitive constant: $p = c$ **and** $\widehat{\tau} = I_\ell$**.** We have

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \{(c, (c)_\ell)\} \qquad \Delta \vdash (c)_\ell : I_\ell \text{ (MemTPrimitive)}$$

According to the fragment coherence criterion of agreement between τ and $\widehat{\tau}_\star$, we necessarily have $\tau = I_\ell$, hence $\nu = c'$ and $\widehat{\nu} = (c')_\ell$. Finally, $c \rhd c'$ implies $c = c'$, hence $(c)_\ell \blacktriangleright (c')_\ell$ (MConstant rule).

Type variable: $\widehat{\tau} = t \in$ ***TyVars*****.** Suppose that the result holds for $(p, \Delta(t))$. From the definitions of memory typing and of *reprs*, we immediately have

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \textbf{pat2mem}_\Delta(p, \Delta(t)) \qquad \Delta, \varsigma \vdash \widehat{\nu} : \Delta(t) \qquad \Delta, \varsigma \vdash \widehat{\nu} \text{ reprs } (\nu : \tau \text{ as } \Delta(t) \equiv \widehat{\tau}_\star.\widehat{\pi})$$

and our result is immediate from the induction hypothesis.

Constant word: $\widehat{\tau} = (c)_\ell$**.** Since $\widehat{\nu}$ is of type $\widehat{\tau}$, we necessarily have $\widehat{\nu} = (c)_\ell$ and we conclude with

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \{(p, (c)_\ell)\} \qquad \Delta \vdash (c)_\ell : (c)_\ell \text{ (MemTConstant)} \qquad \varsigma \vdash (c)_\ell \blacktriangleright (c)_\ell \text{ (MConstant)}$$

The same reasoning applies if $\widehat{\tau}$ is an empty word type $\__\ell$.

Struct: $\widehat{\tau} = \{\!\{\widehat{\tau}_0, \widehat{\tau}_1\}\!\}$**.** Suppose that the result holds for $(p, \widehat{\tau}_0)$ and $(p, \widehat{\tau}_1)$. We have

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \left\{ (p_0 \sqcap p_1, \{\!\{\widehat{p}_0, \widehat{p}_1\}\!\}) \;\middle|\; \begin{array}{c} (p_0, \widehat{p}_0) \in \textbf{pat2mem}_\Delta(p, \widehat{\tau}_0) \\ (p_1, \widehat{p}_1) \in \textbf{pat2mem}_\Delta(p, \widehat{\tau}_1) \\ p_0 \sqcap p_1 \text{ is defined} \end{array} \right\} \qquad \widehat{\nu} = \{\!\{\widehat{\nu}_0, \widehat{\nu}_1\}\!\}$$

$$\Delta, \varsigma \vdash \widehat{\nu}_i : \widehat{\tau}_i \qquad \Delta, \varsigma \vdash \widehat{\nu}_i \text{ reprs } (\nu : \tau \text{ as } \widehat{\tau}_i \equiv \widehat{\tau}_\star.\widehat{\pi}.i)$$

According to our induction hypotheses, for both fields $i \in \{0, 1\}$, there exists $(p_i, \widehat{p}_i) \in \textbf{pat2mem}_\Delta(p, \widehat{\tau}_i)$ such that

$$p_i \rhd \nu \qquad \Delta \vdash \widehat{p}_i : \widehat{\tau}_i \qquad \varsigma \vdash \widehat{p}_i \blacktriangleright \widehat{\nu}_i$$

Since p_0 and p_1 both match the same value ν, they are compatible and their intersection also matches ν: $p_0 \sqcap p_1 \rhd \nu$. We conclude with

$$\frac{\Delta \vdash \widehat{p}_i : \widehat{\tau}_i}{\Delta \vdash \{\!\{\widehat{p}_0, \widehat{p}_1\}\!\} : \{\!\{\widehat{\tau}_0, \widehat{\tau}_1\}\!\}} \text{ (MemTStruct)} \qquad \frac{\varsigma \vdash \widehat{p}_i \blacktriangleright \widehat{\nu}_i}{\varsigma \vdash \{\!\{\widehat{p}_0, \widehat{p}_1\}\!\} \blacktriangleright \widehat{\nu}} \text{ (MStruct)}$$

The same reasoning applies if $\widehat{\tau}$ is a struct with any other number of fields, a pointer or a composite word type.

Fragment: $\widehat{\tau} = (\pi \textbf{ as } \widehat{\tau}')$**.** According to the fragment coherence criterion between τ and $\widehat{\tau}_\star$, since $\widehat{\tau}$ is a fragment that appears within $\widehat{\tau}_\star$, $\tau' = \textbf{focus}_\Delta(\pi, \tau)$ is defined and agrees with $\widehat{\tau}'$. Let $p' = \textbf{focus}(\pi, p)$ and $\nu' = \textbf{focus}(\pi, \nu)$. Suppose that the result holds for $(p', \widehat{\tau}')$. We have

$$\textbf{pat2mem}_\Delta(p, \widehat{\tau}) = \textbf{pat2mem}_\Delta(p', \widehat{\tau}') \qquad p' \rhd \nu' \qquad \Delta, \varsigma \vdash \widehat{\nu} : \widehat{\tau}' \qquad \Delta, \varsigma \vdash \widehat{\nu} \text{ reprs } (\nu' : \tau' \text{ as } \widehat{\tau}' \equiv \widehat{\tau}'.\varepsilon)$$

According to our induction hypothesis, there exists $(p'', \widehat{p}) \in \textbf{pat2mem}_\Delta(p', \widehat{\tau}')$ such that

$$p'' \rhd \nu' \qquad \Delta \vdash \widehat{p} : \widehat{\tau}' \qquad \varsigma \vdash \widehat{p} \blacktriangleright \widehat{\nu}$$

and we conclude with

$$\frac{\Delta \vdash \widehat{p} : \widehat{\tau}'}{\Delta \vdash \widehat{p} : \widehat{\tau}} \text{ (MemTFragment)}$$

Split: $\widehat{\tau} = \mathbf{split}(\dots)$. Let $\{(p_i, \widehat{\tau}_i) \mid 0 \leqslant i < n\} = \widehat{\tau}/\_$. According to the branch coherence criterion between τ and $\widehat{\tau}_\star$, since $\widehat{\tau}$ is a split that appears within $\widehat{\tau}_\star$, there exists a branch $i \in \{0, \dots, n-1\}$ such that $p_i \rhd v$. Since p also matches v, the pattern intersection $p' = p \sqcap p_i$ is defined and matches v. Let $\tau' = \tau/p'$ and $\widehat{\tau}'_\star = \widehat{\tau}_\star[.\widehat{\pi} \leftarrow \widehat{\tau}_i]$. Suppose that the result holds for $(p', \widehat{\tau}_i)$. We have

$$\mathbf{pat2mem}_\Delta(p', \widehat{\tau}_i) \subseteq \mathbf{pat2mem}_\Delta(p, \widehat{\tau}) \qquad p' \rhd v \qquad \Delta \vdash p' : \tau' \qquad \Delta \vdash v : \tau' \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}'$$

$$\Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau' \text{ as } \widehat{\tau}' \equiv \widehat{\tau}'_\star.\widehat{\pi})$$

and according to our induction hypothesis, there exists $(p'', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p', \widehat{\tau}_i)$ such that

$$p'' \rhd v \qquad \Delta \vdash \widehat{p} : \widehat{\tau}_i \qquad \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$$

We conclude with

$$\frac{\Delta \vdash \widehat{p} : \widehat{\tau}_i}{\Delta \vdash \widehat{p} : \widehat{\tau}} \text{ (MemTSplit)}$$

□

Lemma 3.12 (Memory matching implies high-level matching). *Let* $\Delta, \varsigma, \tau, \widehat{\tau}, p, p', v, \widehat{p}$ *and* $\widehat{v}$ *such that*

$$\vDash \Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon) \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau} \qquad \Delta \vdash p : \tau \qquad \Delta \vdash \widehat{p} : \widehat{\tau} \qquad (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau})$$

$$\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$$

We have $p \rhd v$.

Proof. Let us proceed by induction on $(p, \widehat{\tau})$.

Wildcard base case: $p = \_$. Immediate:

$$\mathbf{pat2mem}_\Delta(\_, \widehat{\tau}) = \{(\_, \__{|\widehat{\tau}|})\} \qquad \_ \rhd v$$

Primitive bit constant base case: $p = c$ **and** $\widehat{\tau} = I_1$. We have $\mathbf{pat2mem}_\Delta(c, I_1) = \{(c, (c)_1)\}$, hence $\widehat{p} = (c)_1$. According to the fragment coherence criterion of agreement between τ and $\widehat{\tau}$, we necessarily have $\tau = I_1$ (or a type variable which we unroll to I_1), hence $v = c'$ and $\widehat{v} = (c')_1$. Finally, since $\varsigma \vdash (c)_1 \blacktriangleright (c')_1$, we have $c = c'$ (MConstant rule), hence $c \rhd c'$.

Induction step. Here, we finally use the various intermediate results stated earlier. According to Lemma 3.6, since $\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$, there exists a branch $(p_b, \widehat{\tau}_b) \in \widehat{\tau}/\_$ such that

$$\Delta \vdash \widehat{p} : \widehat{\tau}_b \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}_b \qquad \forall (\widehat{\pi} \mapsto \pi \text{ as } \widehat{\tau}') \in \mathbf{shatter}_\Delta(\widehat{\tau}_b), \varsigma \vdash \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{p}) \blacktriangleright \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \widehat{v})$$

Let $\tau_b = \tau/p_b$. From Lemmas 3.7 and 3.9, we have

$$p_b \sqcap p' \text{ is defined} \qquad (p', \widehat{p}) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}_b) \qquad p_b \rhd v \qquad \Delta \vdash v : \tau_b$$

$$\Delta, \varsigma \vdash \widehat{v} \text{ reprs } (v : \tau_b \text{ as } \widehat{\tau}_b \equiv \widehat{\tau}_b.\varepsilon)$$

Induction hypothesis: suppose that for every $(\widehat{\pi} \mapsto \pi \text{ as } \widehat{\tau}') \in \mathbf{shatter}_\Delta(\widehat{\tau}_b)$, the result holds for $(\mathbf{focus}\,(\pi, p), \widehat{\tau}')$. We use Lemma 3.5 to show that $p \rhd v$.

Each individual bit of every primitive matches: Let $\pi \in \mathit{Paths}$ such that

$$\mathbf{focus}_\Delta\,(\pi, \tau) = I_1 \qquad \mathbf{focus}\,(\pi, p) = c \qquad \mathbf{focus}\,(\pi, v) = c'$$

We show that $c = c'$. According to the coverage criterion of agreement between τ and $\widehat{\tau}$, there exist π_f, π', $\widehat{\pi}$ and $\widehat{\tau}_f$ such that $\pi = \pi_f.\pi'$ and $(\widehat{\pi} \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathbf{shatter}_\Delta(\widehat{\tau}_b)$. Let $\widehat{\tau}' = \widehat{\mathbf{focus}}_\Delta\,(\widehat{\pi}, \widehat{\tau}_b)$, $\widehat{p}_f = \widehat{\mathbf{focus}}\,(\widehat{\pi}, \widehat{p})$ and $\widehat{v}_f = \widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi}, \widehat{v})$. We have:

$$\varsigma \vdash \widehat{p}_f \blacktriangleright \widehat{v}_f \qquad \exists p'', (p'', \widehat{p}_f) \in \mathbf{pat2mem}_\Delta(p, \widehat{\tau}') \text{ (Lemma 3.8)}$$

$$\Delta, \varsigma \vdash \widehat{v}_f \text{ reprs } (v : \tau_b \text{ as } \widehat{\tau}' \equiv \widehat{\tau}_b.\widehat{\pi}) \text{ (Lemma 3.10)}$$

Let $\tau_f = \mathbf{focus}_\Delta(\pi_f, \tau_b)$, $p_f = \mathbf{focus}(\pi_f, p)$, $p'_f = \mathbf{focus}(\pi_f, p'')$ and $v_f = \mathbf{focus}(\pi_f, v)$. Since $\widehat{\tau}'$ is a fragment or primitive type representing the piece of data at position π_f as $\widehat{\tau}_f$, from the definitions of **pat2mem** and *reprs*, we have:

$$(p'_f, \widehat{p}_f) \in \mathbf{pat2mem}_\Delta(p_f, \widehat{\tau}_f) \qquad \Delta, \varsigma \vdash \widehat{v}_f \text{ reprs } (v_f : \tau_f \text{ as } \widehat{\tau}_f \equiv \widehat{\tau}_f.\varepsilon)$$

According to our induction hypothesis, we have $p_f \rhd v_f$, therefore $\mathbf{focus}(\pi', p_f) \rhd \mathbf{focus}(\pi', v_f)$, and we conclude that $c = c'$ with

$$\mathbf{focus}(\pi', p_f) = \mathbf{focus}(\pi, p) = c \qquad \mathbf{focus}(\pi', v_f) = \mathbf{focus}(\pi, v) = c' \qquad c \rhd c'$$

Each head constructor matches: Let $\pi \in \mathit{Paths}$ such that

$$\mathbf{focus}_\Delta(\pi, \tau) = K_0(\tau_0) \mid \cdots \mid K_{n-1}(\tau_{n-1}) \qquad \mathbf{focus}(\pi, p) = K_i(p'') \qquad \mathbf{focus}(\pi, v) = K_j(v')$$

We prove by contradiction that $K_i = K_j$. Suppose that $K_i \neq K_j$. According to the branch coherence criterion of agreement between τ and $\widehat{\tau}$, τ_b agrees with $\widehat{\tau}_b$. According to the distinguishability criterion of agreement between τ_b and $\widehat{\tau}_b$ and because $\widehat{\tau}_b$ is already a specialized memory type, there exists a memory path $\widehat{\pi}$ such that either $\widehat{\mathbf{focus}}_\Delta(\widehat{\pi}, \widehat{\tau}_b) = (c_i)_\ell = (c_j)_\ell$ with $c_i \neq c_j$ – which is impossible – or there exists π_f, π' and $\widehat{\tau}_f$ such that $\pi = \pi_f.\pi'$ and $(\widehat{\pi} \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathbf{shatter}_\Delta(\widehat{\tau}_b)$. Let $\tau_f = \mathbf{focus}_\Delta(\pi_f, \tau_b)$, $p_f = \mathbf{focus}(\pi_f, p)$, $p'_f = \mathbf{focus}(\pi_f, p'')$ and $v_f = \mathbf{focus}(\pi_f, v)$. Since $\widehat{\tau}'$ is a fragment or primitive type representing the piece of data at position π_f as $\widehat{\tau}_f$, from the definitions of **pat2mem** and *reprs*, we have:

$$(p'_f, \widehat{p}_f) \in \mathbf{pat2mem}_\Delta(p_f, \widehat{\tau}_f) \qquad \Delta, \varsigma \vdash \widehat{v}_f \text{ reprs } (v_f : \tau_f \text{ as } \widehat{\tau}_f \equiv \widehat{\tau}_f.\varepsilon)$$

According to our induction hypothesis, we have $p_f \rhd v_f$, therefore $\mathbf{focus}(\pi', p_f) \rhd \mathbf{focus}(\pi', v_f)$. Since $\mathbf{focus}(\pi', p_f) = \mathbf{focus}(\pi, p) = K_i(p'')$ and $\mathbf{focus}(\pi', v_f) = \mathbf{focus}(\pi, v) = K_j(v')$, this implies $K_i = K_j$.

□

3.4.4 Results on $\hookrightarrow$ and $\twoheadrightarrow$

We are now ready to state and prove our main results. Most of our proofs proceed by induction on memory types, or on $\mathcal{R}$ (defined in Fig. 3.32) derivation trees. Even though types may be recursive, expressions are finite, ensuring that typing and representation derivation trees are always finite.

Lemma 3.13 ($\twoheadrightarrow_m$ preserves $\mathcal{R}$ on memory valuexpressions)**.** *Let* $\Delta, \Sigma, \tau, \widehat{\tau}, S = (\Gamma_h, \sigma_h, (u_h : \tau_h \text{ as } \widehat{\tau}_h))$, $\widehat{S} = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{u})$ *and* $\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{u}')$ *(note that* $\widehat{u}, \widehat{u}' \in \widehat{\mathit{ValuExprs}}$*) such that*

$$\Delta, \Sigma, \tau \text{ as } \widehat{\tau} \vdash S \mathcal{R} \widehat{S} \qquad \Delta \vdash \widehat{S} \twoheadrightarrow_m \widehat{S}'$$

We have

$$\Delta, \Sigma, \tau \text{ as } \widehat{\tau} \vdash S \mathcal{R} \widehat{S}'$$

Proof. From now on, we omit Δ and Σ from judgments. Let $\sigma = \sigma_h \sqcup \sigma_m$, $\sigma' = \sigma_h \sqcup \sigma'_m$, and $v_h = u_h[\![\sigma]\!]$. Note that since $\twoheadrightarrow_m$ does not affect existing bindings in σ, we always have $u_h[\![\sigma']\!] = v_h$.

Without loss of generality, we assume that $\tau_h = \tau$ and $\widehat{\tau}_h = \widehat{\tau}$. Indeed, our typing hypothesis $\Gamma_h \vdash (u_h : \tau_h \text{ as } \widehat{\tau}_h) : \tau \text{ as } \widehat{\tau}$ necessarily involves the following TPivot step:

$$\textsc{TPivot}\ \frac{\vDash \widehat{\tau}_h \qquad \mathbf{agree}(\tau_h, \widehat{\tau}_h) \qquad \Gamma_h \vdash u_h : \tau_h}{\Gamma_h \vdash (u_h : \tau_h \text{ as } \widehat{\tau}_h) : \tau_h \text{ as } \widehat{\tau}_h}$$

after which only TMTyVar, MemTTypeVar, MemTSplit and MemTFragment rules are applicable to reach the type pair τ as $\widehat{\tau}$. – meaning that $\widehat{\tau}$ is essentially a more general version of $\widehat{\tau}_h$.

Some preconditions of our goal τ as $\widehat{\tau} \vdash S \mathrel{\mathcal{R}} \widehat{S'}$ are unchanged from our hypothesis τ as $\widehat{\tau} \vdash S \mathrel{\mathcal{R}} \widehat{S}$. We only need to prove the four following properties to prove preservation:

$$\vDash \Gamma'_m, \varsigma' \vdash \widehat{\sigma'} \tag{3.1}$$

$$\Gamma'_m, \varsigma' \vdash \widehat{\sigma'} \text{ reprs } \sigma' \tag{3.2}$$

$$\Gamma'_m, \varsigma' \vdash \widehat{u'} : \widehat{\tau} \tag{3.3}$$

$$\varsigma' \vdash \widehat{u'}[\![\sigma']\!] \text{ reprs } (v_h : \tau_h \text{ as } \widehat{\tau}_h \equiv \widehat{\tau}_h.\varepsilon) \tag{3.4}$$

We proceed by induction on $\widehat{u}$. For each case, we show that every possible $\mapsto_m$ step preserves the relation. Most $\mapsto_m$ rules apply to pivots, which are our main base case.

Pivot: $\widehat{u} = (u_m : \tau_m \textbf{ as } \widehat{\tau}_m \equiv \widehat{\tau}_\star.\widehat{\pi})$ (and $\widehat{\tau}_m$ is not a type variable). Without loss of generality, we assume that $\tau_m = \tau$ and $\widehat{\tau}_m = \widehat{\tau}$, using the same reasoning as above for $\tau_h = \tau$ and $\widehat{\tau}_h = \widehat{\tau}$. Our typing hypothesis becomes:

$$\Gamma_m, \varsigma \vdash (u_m : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) : \widehat{\tau}$$

and implies (since we have to use the TPivot and MemTHLExp rules):

$$\textbf{agree}(\tau, \widehat{\tau}_\star) \qquad \Gamma_m \vdash u_m : \tau \qquad \widehat{\textbf{focus}}\,(\widehat{\pi}, \widehat{\tau}_\star) = \widehat{\tau}$$

Using a similar reasoning, we also assume that $u_m[\![\sigma]\!] = v_h$. Indeed, the representation relation between two pivots is restricted to RIdentity and to RTypeVar, RFragment and RSplit, which do not depend on the pivot value. Our representation hypothesis becomes:

$$\varsigma \vdash (v_h : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) \text{ reprs } (v_h : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)$$

Let us proceed by case analysis on the $\mapsto_m$ rules. We start with subterm extraction rules.

EVarAccess: there exists $(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma_m$ such that

$$u_m = x.\varepsilon \qquad \widehat{\tau}_x = \widehat{\tau} \qquad \widehat{S'} = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{\sigma}(x))$$

We prove our result for $\widehat{S'}$:

Eq. (3.1) and Eq. (3.2) are immediate from preconditions.

Eq. (3.3): $\Gamma_m, \varsigma \vdash \widehat{\sigma}(x) : \widehat{\tau}$ follows from the environment typing hypothesis $\Gamma_m, \varsigma \vdash \widehat{\sigma}$.

Eq. (3.4): $\varsigma \vdash \widehat{\sigma}(x)$ reprs $(v_h : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)$.
Since we have $v_h = x.\varepsilon[\![\sigma]\!] = \sigma(x)[\![\sigma]\!]$, this is implied by our environment representation hypothesis $\Gamma_m, \varsigma \vdash \widehat{\sigma}$ reprs σ.

EVarFocus: there exist

$$(x : \tau_x \text{ as } \widehat{\tau}_x) \in \Gamma_m \qquad \pi \in \mathit{Paths} \qquad (p_b, \widehat{\tau}_b) \in \widehat{\tau}_x /\!\_ \qquad (\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \textbf{shatter}(\widehat{\tau}_b)$$

$$x_f \in \mathit{Vars} \setminus \operatorname{dom}(\Gamma_m)$$

such that

$$u_m = x.(\pi_f.\pi) \qquad \Gamma_m, \varsigma \vdash \widehat{\sigma}(x) : \widehat{\tau}_b \qquad \Gamma'_m = \Gamma_m \cup \{(x_f : \textbf{focus}\,(\pi_f, \tau_x) \text{ as } \widehat{\tau}_f)\}$$

$$\sigma'_m = \sigma_m \cup \{x_f \mapsto x.\pi_f\} \qquad \widehat{\sigma'} = \widehat{\sigma} \cup \{x_f \mapsto \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x))\} \qquad \varsigma' = \varsigma$$

$$\widehat{u'} = (x_f.\pi : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$$

We prove our result for $\widehat{S'}$:

Eq. (3.1): $\vDash \widehat{\tau}_f$, $\textbf{agree}(\textbf{focus}\,(\pi_f, \tau_x), \widehat{\tau}_f)$ and $\Gamma'_m, \varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) : \textbf{focus}\,(\pi_f, \tau_x)$ as $\widehat{\tau}_f$.
$\vDash \widehat{\tau}_f$ follows from $\vDash \widehat{\tau}_x$ (implied by our environment typing hypothesis); $\textbf{agree}(\textbf{focus}\,(\pi_f, \tau_x), \widehat{\tau}_f)$ follows from the fragment coherence agreement criterion between τ_x and $\widehat{\tau}_x$, which is also implied by our environment typing hypothesis. The last result is immediate from our environment typing hypothesis, which implies $\Gamma_m, \varsigma \vdash \widehat{\sigma}(x) : \tau_x$ as $\widehat{\tau}_x$, using structural and fragment or primitive memory typing rules to descend into the memory value, high-level type and memory type.

Eq. (3.2): $\varsigma' \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x))$ reprs $(x.\pi_f[\![\sigma]\!] : \textbf{focus}\,(\pi_f, \tau_x)$ as $\widehat{\tau}_f \equiv \widehat{\tau}_f.\varepsilon)$.
Our environment representation hypothesis implies:

$$\varsigma \vdash \widehat{\sigma}(x) \text{ reprs } (x.\varepsilon[\![\sigma]\!] : \tau_x \text{ as } \widehat{\tau}_x \equiv \widehat{\tau}_x.\varepsilon)$$

We destruct the *reprs* derivation tree leading to this conclusion, and show that it necessarily involves a rule whose conditions lead to our result. Since $\widehat{\sigma}(x)$ is a memory value, the rules RIdentity and RAddress will never be used. Starting from the conclusion and going backwards in *reprs* deduction steps, we go through structural rules before encountering the first split in $\widehat{\tau}_x$ at some position $\widehat{\pi}_b$, which is derived from the following RSplit step:

$$\textsc{RSplit}\ \frac{\vdash x.\varepsilon[\![\sigma]\!] : \tau_x \,/\, p_b \qquad \varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_b, \widehat{\sigma}(x)) \text{ reprs } (x.\varepsilon[\![\sigma]\!] : \tau_x \,/\, p_b \text{ as } \widehat{\tau}_b \equiv \widehat{\tau}_x\,[.\widehat{\pi}_b \leftarrow \widehat{\tau}_b]\,.\widehat{\pi}_b)}{\varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_b, \widehat{\sigma}(x)) \text{ reprs } (x.\varepsilon[\![\sigma]\!] : \tau_x \text{ as } \widehat{\textbf{focus}}\,(\widehat{\pi}_b, \widehat{\tau}_x) \equiv \widehat{\tau}_x.\widehat{\pi}_b)}$$

From there, we continue to descend into the memory type until we reach the position $\widehat{\pi}_f$. We have $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \textbf{shatter}(\widehat{\tau}_b)$, which means that $\widehat{\textbf{focus}}\,(\widehat{\pi}_f, \widehat{\tau}_b)$ is either a fragment or a primitive type. If it is a fragment, we go through the following RFragment step:

$$\textsc{RFragment}\ \frac{\varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) \text{ reprs } (\textbf{focus}\,(\pi_f, x.\varepsilon[\![\sigma]\!]) : \textbf{focus}\,(\pi_f, \tau_x \,/\, p_b) \text{ as } \widehat{\tau}_f \equiv \widehat{\tau}_f.\varepsilon)}{\varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) \text{ reprs } (x.\varepsilon[\![\sigma]\!] : \tau_x \,/\, p_b \text{ as } (\pi_f \text{ as } \widehat{\tau}_f) \equiv \widehat{\tau}_x\,[.\widehat{\pi}_b \leftarrow \widehat{\tau}_b]\,.\widehat{\pi}_f)}$$

and our result is immediate from the precondition of this step. Otherwise, it is a primitive type: $\widehat{\tau}_f = I_\ell$ and $\pi_f = \varepsilon$. We go through either a RAtom or a RFission rule. Here, we only cover the RAtom case (the RFission case is similar to a combination of RFragment and RAtom cases). This rule does not have any preconditions, but restricts the possible shapes for our expressions:

$$\textsc{RAtom}\ \frac{\widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) = (c)_\ell \qquad x.\varepsilon[\![\sigma]\!] = c}{\varsigma \vdash \widehat{\textbf{focus}}_\varsigma\,(\widehat{\pi}_f, \widehat{\sigma}(x)) \text{ reprs } (x.\varepsilon[\![\sigma]\!] : \tau_x \,/\, p_b \text{ as } I_\ell \equiv \widehat{\tau}_x\,[.\widehat{\pi}_b \leftarrow \widehat{\tau}_b]\,.\widehat{\pi}_f)}$$

We use these constraints to prove our result using the RAtom rule:

$$\textsc{RAtom}\ \varsigma \vdash (c)_\ell \text{ reprs } (c : \tau_x \text{ as } I_\ell \equiv I_\ell.\varepsilon)$$

Eq. (3.3): $\Gamma'_m, \varsigma' \vdash (x_f.\pi : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi}) : \widehat{\tau}$.
We only have to prove $\Gamma'_m \vdash x_f.\pi : \tau$, which is immediate from the definition of Γ'_m.

Eq. (3.4): $\varsigma' \vdash (x_f.\pi[\![\sigma']\!] : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}_\star.\widehat{\pi})$ reprs $(v_h : \tau \text{ as } \widehat{\tau} \equiv \widehat{\tau}.\varepsilon)$ is immediate from our representation hypothesis since $x_f.\pi[\![\sigma']\!] = x.\pi_f.\pi[\![\sigma]\!] = v_h$.

Other rules are specific to a given memory type – either a primitive type, a synchronization point with the high-level type or a memory structure. We destruct $\widehat{\tau}_m$ and prove the result for each case.

EAtom: we have

$$v_h = c \qquad \widehat{\tau} = I_\ell \qquad \widehat{S'} = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, (c)_\ell)$$

Our result is immediate: we do have $\Gamma_m, \varsigma \vdash (c)_\ell : I_\ell$ and $\varsigma \vdash (c)_\ell$ reprs $(c : \tau \text{ as } I_\ell \equiv I_\ell.\varepsilon)$.

All other rules: similar to EAtom, each evaluation rule restricts $\widehat{u}'$, τ, $\widehat{\tau}$ and v_h so that our result is immediate from the corresponding typing and representation rules.

Type variable pivot: $\widehat{u} = (u_m : \tau_m \textbf{ as } t_m \equiv \widehat{\tau}_\star.\widehat{\pi})$ with $t_m \in \mathit{TyVars}$. The only applicable $\twoheadrightarrow_m$ rules are subterm extraction rules, which we treated in the previous pivot case, and ETypeVar. Suppose that the property holds for $(u_m : \tau_m \text{ as } \Delta(t_m) \equiv \widehat{\tau}_\star.\widehat{\pi})$. In the ETypeVar case, we have:

$$(\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, (u_m : \tau_m \text{ as } \Delta(t_m) \equiv \widehat{\tau}_\star.\widehat{\pi}) \twoheadrightarrow_m \widehat{S'}$$

and our result is immediate from the induction hypothesis.

Pointer expression: $\widehat{u} = \&_\ell(\widehat{v})$**.** The only applicable $\rightsquigarrow_m$ rule is EAddress. We have

$$\widehat{S}' = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma \cup \{a \mapsto \widehat{v}\}, \&_\ell(a))$$

with $a \notin \mathrm{dom}(\varsigma)$, and our result is immediate using MemTAddress and RAddress rules.

Other expression: $\widehat{u} = C[\widehat{u}_0]$ where C is a memory context. The only applicable $\rightsquigarrow_m$ rule is ESubStep. Without loss of generality, we assume that there exists a memory type $\widehat{\tau}_0$ such that $\widehat{\tau} = C[\widehat{\tau}_0]$. Suppose that the result holds for $\tau, \widehat{\tau}_0, S_0 = (\Gamma_h, \sigma_h, (u_h : \tau \text{ as } \widehat{\tau}_0))$ and $\widehat{S}_0 = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{u}_0)$. The precondition τ as $\widehat{\tau}_0 \vdash S_0 \mathcal{R} \widehat{S}_0$ is immediate from τ as $\widehat{\tau} \vdash S \mathcal{R} \widehat{S}$. We have

$$\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', C[\widehat{u}'_0]) \qquad \widehat{S}_0 \rightsquigarrow_m \widehat{S}'_0 = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{u}'_0)$$

From the induction hypothesis, we get τ as $\widehat{\tau}_0 \vdash S_0 \mathcal{R} \widehat{S}'_0$, from which our conclusion τ as $\widehat{\tau} \vdash S \mathcal{R} \widehat{S}'$ is immediate.

□

Lemma 3.14 ($\rightsquigarrow_m$ progresses on memory valuexpressions)**.** *Let* Δ, $\widehat{\tau}$ *and* $\widehat{S} = (\Gamma, \sigma, \widehat{\sigma}, \varsigma, \widehat{u})$ *(note that* $\widehat{u} \in \widehat{\mathit{ValuExprs}}$*) such that*

$$\Delta \vDash \widehat{\tau} \qquad \vDash \Delta, \Gamma, \varsigma \vdash \widehat{\sigma} \qquad \vDash \Delta, \Gamma, \varsigma \vdash \widehat{u} : \widehat{\tau}$$

There exists at least one state $\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{u}')$ *such that* $\Delta \vdash \widehat{S} \rightsquigarrow_m \widehat{S}'$*.*

Proof. Immediate by induction on $\widehat{u}$. □

Lemma 3.15 ($\rightsquigarrow$ simulates $\hookrightarrow$)**.** *Let* Δ, Σ, τ, $\widehat{\tau}$, $S = (\Gamma_h, \sigma_h, e)$ *and* $\widehat{S} = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{e})$ *such that*

$$\Delta, \Sigma, \tau \text{ as } \widehat{\tau} \vdash S \mathcal{R} \widehat{S}$$

From now on, we omit Δ *and* Σ *from judgments. One of the three following conditions holds:*

- *Both expressions are in normal form:* $e = (u : \tau \text{ as } \widehat{\tau})$ *and* $\widehat{e} \in \widehat{\mathit{Values}}$*;*
- $\widehat{S}$ *steps and remains bisimilar to* S*: there exists* $\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{e}')$ *such that* $\widehat{S} \rightsquigarrow_m \widehat{S}'$ *and* $(\tau \text{ as } \widehat{\tau}) \vdash S \mathcal{R} \widehat{S}'$*;*
- *both* S *and* $\widehat{S}$ *step and remain bisimilar to each other: there exist* $S' = (\Gamma'_h, \sigma'_h, e')$ *and* $\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{e}')$ *such that* $S \hookrightarrow S'$, $\widehat{S} \rightsquigarrow_h \widehat{S}'$ *and* $(\tau \text{ as } \widehat{\tau}) \vdash S' \mathcal{R} \widehat{S}'$*.*

Proof. Let $\sigma = \sigma_h \sqcup \sigma_m$. We proceed by induction on $\Gamma_m, \sigma, \varsigma \vdash e \mathcal{R} \widehat{e}$:

Function call: $e = \widehat{e} = f(x)$**.** Since f(x) is of type $(\tau \text{ as } \widehat{\tau})$, there exist τ' and $\widehat{\tau}'$ such that $(x : \tau' \text{ as } \widehat{\tau}') \in \Gamma_h$ and $(f : \tau' \text{ as } \widehat{\tau}' \to \tau \text{ as } \widehat{\tau}) \in \Gamma_h$. Let x' and e' such that $\Sigma(f) = \lambda x'.e'$. We assume that x' is a unique symbol. We have

$$S \hookrightarrow_{\text{HLEFunCall}} S' = (\Gamma'_h, \sigma'_h, e') \qquad \widehat{S} \hookrightarrow_{\text{hEFunCall}} \widehat{S}' = (\Gamma'_m, \sigma_m, \widehat{\sigma}', \varsigma, e')$$

where

$$\Gamma'_h = \Gamma_h \cup \{(x' : \tau' \text{ as } \widehat{\tau}')\} \qquad \sigma'_h = \sigma_h \cup \{x' \mapsto \sigma_h(x)\} \qquad \Gamma'_m = \Gamma_m \cup \{(x' : \tau' \text{ as } \widehat{\tau}')\}$$

$$\widehat{\sigma}' = \widehat{\sigma} \cup \{x' \mapsto \widehat{\sigma}(x)\}$$

Using hypotheses on S and $\widehat{S}$, we immediately have τ as $\widehat{\tau} \vdash S' \mathcal{R} \widehat{S}'$.

Pattern matching: $e = \widehat{e} = \mathsf{match}(x)\left\{ p_i \;\to\; e_i \;\middle|\; 0 \leqslant i < n \right\}$. Let $i \in \{0, \ldots, n-1\}$ the smallest branch such that $\sigma_h \vdash p_i \rhd \sigma_h(x)$. Since e is well-typed, such a branch always exists. Let τ' and $\widehat{\tau}'$ such that $(x : \tau' \text{ as } \widehat{\tau}') \in \Gamma_h$. According to Theorem 3.2, i is also the first branch that matches $\widehat{\sigma}(x)$ at the memory level, that is, the smallest $i \in \{0, \ldots, n-1\}$ for which there exists $(p', \widehat{p}) \in \textbf{pat2mem}(p_i, \widehat{\tau}')$ such that $\varsigma \vdash \widehat{p} \blacktriangleright \widehat{\sigma}(x)$. We have

$$S \hookrightarrow_{\text{HLEMatch}} S' = (\Gamma_h, \sigma_h, e_i) \qquad \widehat{S} \hookrightarrow_{\text{hEMatch}} \widehat{S}' = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, e_i)$$

and τ as $\widehat{\tau} \vdash S' \mathcal{R} \widehat{S}'$ is immediate from hypotheses on S and $\widehat{S}$.

Let-binding: there exist x, τ', $\widehat{\tau}'$, e_0, e' and $\widehat{e}'$ such that

$$e = \mathsf{let}\ x : \tau' \text{ as } \widehat{\tau}' = e'\ \mathsf{in}\ e_0 \qquad \widehat{e} = \mathsf{let}\ x : \tau' \text{ as } \widehat{\tau}' = \widehat{e}'\ \mathsf{in}\ e_0 \qquad \Gamma_m, \sigma, \varsigma \vdash e' \mathcal{R} \widehat{e}'$$

Let $S' = (\Gamma_h, \sigma_h, e')$ and $\widehat{S}' = (\Gamma_m, \sigma_m, \widehat{\sigma}, \varsigma, \widehat{e}')$. Since e and $\widehat{e}$ are well-typed, and using hypotheses on S and $\widehat{S}$, we have τ' as $\widehat{\tau}' \vdash S' \mathcal{R} \widehat{S}'$ and use the induction hypothesis:

- If both bound expressions are in normal form, i.e., $e' = (u' : \tau' \text{ as } \widehat{\tau}')$ and $\widehat{e}' = \widehat{v} \in \widehat{\mathit{Values}}$, then both expressions go through a let-binding step: we have

  $$S \hookrightarrow_{\text{HLELetBind}} S_0 = (\Gamma'_h, \sigma'_h, e_0) \qquad \widehat{S} \hookrightarrow_{\text{hELetBind}} \widehat{S}_0 = (\Gamma'_m, \sigma_m, \widehat{\sigma}', \varsigma, e_0)$$

  where

  $$\Gamma'_h = \Gamma_h \cup \{(x : \tau' \text{ as } \widehat{\tau}')\} \qquad \sigma'_h = \sigma_h \cup \{x \mapsto u'\} \qquad \Gamma'_m = \Gamma_m \cup \{(x : \tau' \text{ as } \widehat{\tau}')\}$$
  $$\widehat{\sigma}' = \widehat{\sigma} \cup \{x \mapsto \widehat{v}\}$$

  and τ as $\widehat{\tau} \vdash S_0 \mathcal{R} \widehat{S}_0$ is immediate from hypotheses on S and $\widehat{S}$.
- If $\widehat{S}'$ steps and remains bisimilar to S', i.e., there exists $\widehat{S}'' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{e}'')$ such that $\widehat{S}' \mapsto_m \widehat{S}''$ and τ' as $\widehat{\tau}' \vdash S' \mathcal{R} \widehat{S}''$, then $\widehat{S}$ goes through the same step using the ESubStep rule and remains bisimilar to S: we have

  $$\widehat{S} \mapsto_m \widehat{S}^\star = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \mathsf{let}\ x : \tau' \text{ as } \widehat{\tau}' = \widehat{e}''\ \mathsf{in}\ e_0)$$

  and τ as $\widehat{\tau} \vdash S \mathcal{R} \widehat{S}^\star$ is immediate from hypotheses on S, $\widehat{S}$ and $\widehat{S}''$.
- If both S' and $\widehat{S}'$ step and remain bisimilar, i.e., there exist a rule label l, $S'' = (\Gamma'_h, \sigma'_h, e'')$ and $\widehat{S}'' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{e}'')$ such that $S' \hookrightarrow_l S''$, $\widehat{S}' \mapsto_{hl} \widehat{S}''$ and τ' as $\widehat{\tau}' \vdash S'' \mathcal{R} \widehat{S}''$, then both S and $\widehat{S}$ go through the same step using the (HL)ELetStep rule and remain bisimilar: we have

  $$S \hookrightarrow_l S^\star = (\Gamma'_h, \sigma'_h, \mathsf{let}\ x : \tau' \text{ as } \widehat{\tau}' = e''\ \mathsf{in}\ e_0)$$
  $$\widehat{S} \mapsto_{hl} \widehat{S}^\star = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \mathsf{let}\ x : \tau' \text{ as } \widehat{\tau}' = \widehat{e}''\ \mathsf{in}\ e_0)$$

  and τ as $\widehat{\tau} \vdash S^\star \mathcal{R} \widehat{S}^\star$ is immediate from hypotheses on S, $\widehat{S}$, S'' and $\widehat{S}''$.

Pivot: there exist u, $\widehat{u}$, τ' and $\widehat{\tau}'$ such that

$$e = (u : \tau' \text{ as } \widehat{\tau}') \qquad \widehat{e} = \widehat{u} \qquad \Gamma_m, \sigma, \varsigma \vdash \widehat{u} \text{ reprs } (u : \tau' \text{ as } \widehat{\tau}' \equiv \widehat{\tau}'.\varepsilon)$$

According to Lemma 3.14, there are two possible cases:

- $\widehat{u}$ is in normal form, i.e., $\widehat{u} \in \widehat{\mathit{Values}}$. The first condition holds: e is a pivot and $\widehat{e}$ is a memory value.
- There exists $\widehat{S}' = (\Gamma'_m, \sigma'_m, \widehat{\sigma}', \varsigma', \widehat{u}')$ such that $\widehat{S} \mapsto_m \widehat{S}'$. According to Lemma 3.13, since τ as $\widehat{\tau} \vdash S \mathcal{R} \widehat{S}$, we have τ as $\widehat{\tau} \vdash S \mathcal{R} \widehat{S}'$, therefore the second condition holds.

□

At last, we can state our final branching bisimulation theorem, which formalizes the intuition given by the diagram shown in Fig. 3.31.

Theorem 3.3 ($\mathcal{R}$ is a branching bisimulation)**.** *Let* S *and* $\widehat{S}$ *such that* $S \mathrel{\mathcal{R}} \widehat{S}$*, we have:*

- *if* $S \hookrightarrow_{\mathtt{l}} S'$*, then there exist* $\widehat{S}'$ *and* $\widehat{S}''$ *such that* $\widehat{S} \twoheadrightarrow^{*}_{\mathtt{m}} \widehat{S}' \twoheadrightarrow_{\mathtt{hl}} \widehat{S}''$*,* $S \mathrel{\mathcal{R}} \widehat{S}'$ *and* $S' \mathrel{\mathcal{R}} \widehat{S}''$*;*
- *if* $\widehat{S} \twoheadrightarrow_{\mathtt{m}} \widehat{S}'$*, then* $S \mathrel{\mathcal{R}} \widehat{S}'$*;*
- *if* $\widehat{S} \twoheadrightarrow_{\mathtt{hl}} \widehat{S}'$*, then there exists* S' *such that* $S \hookrightarrow_{\mathtt{l}} S'$ *and* $S' \mathrel{\mathcal{R}} \widehat{S}'$*.*

Proof. Immediate from Lemma 3.15. □

3.5 Conclusion

In this chapter, we have formalized the Ribbitulus, whose syntax includes a formal version of the user-visible language presented in Chapter 2 – ADTs and their inhabitants, and memory types – as well as a model of memory contents. For both high-level and memory-level components of the language, we defined their semantics through a combination of a typing judgment and a small-step evaluation judgment. Perhaps most importantly, we formally defined the notion of *agreement* between high-level and memory types, and used it to show that high-level and memory semantics of a given program are always coherent with each other. Although Ribbit currently only supports programs operating on finite data, the choice of a small-step style to define the semantics of the Ribbitulus independently of program (non-)termination could potentially allow us to consider infinite values (e.g., streams of data) in the future. Of course, this would still require a significant extension to some aspects of its core design, and probably some form of coinductive proofs.

In the next part of this thesis, we provide a formal compilation scheme for the Ribbitulus.

		High-level	Memory-level
Syntax	identifiers	(variables) $x \in \mathit{Vars}$ (type variables) $t \in \mathit{TyVars}$	
		(function symbols) $f \in \mathit{FunVars}$	(addresses) $a \in \mathit{Addrs}$
	expressions and values	$e \in E \subseteq \mathit{Exprs}$ Fig. 3.8 $u \in U \subseteq \mathit{ValuExprs}$ $v \in V \subseteq \mathit{Values} \subsetneq \mathit{ValuExprs}$	$\widehat{e} \in \widehat{E} \subseteq \widehat{\mathit{Exprs}} \supsetneq \widehat{\mathit{ValuExprs}} \cup \mathit{Exprs}$ $\widehat{u} \in \widehat{U} \subseteq \widehat{\mathit{ValuExprs}} \subsetneq \widehat{\mathit{Exprs}}$ Fig. 3.17 $\widehat{v} \in \widehat{V} \subseteq \widehat{\mathit{Values}} \subsetneq \widehat{\mathit{ValuExprs}}$
		(pivot expressions) $(u : \tau \text{ as } \widehat{\tau}) \in \mathit{Exprs} \cap \widehat{\mathit{ValuExprs}}$	
	patterns	$p \in P \subseteq \mathit{Patterns}$ Fig. 3.2	$\widehat{p} \in \widehat{P} \subseteq \widehat{\mathit{Patterns}}$ Fig. 3.18
	paths	$\pi \in \Pi \subseteq \mathit{Paths}$ Fig. 3.5	$\widehat{\pi} \in \widehat{\Pi} \subseteq \widehat{\mathit{Paths}}$ Fig. 3.10
	types	$\tau \in T \subseteq \mathit{Types}$ Fig. 3.1	$\widehat{\tau} \in \widehat{T} \subseteq \widehat{\mathit{Types}}$ Fig. 3.9 (kinds) $\widehat{\kappa} \in \{\mathbf{Word}(\{0,1\}^*), \mathbf{Block}\}$
Envs.	typing	$\Delta : \mathit{TyVars} \to \mathit{Types} \cup \widehat{\mathit{Types}}$	$\Gamma : \mathit{Vars} \to \mathit{Types} \times \widehat{\mathit{Types}}$
	value	$\sigma : \mathit{Vars} \to \mathit{ValuExprs}$	$\widehat{\sigma} : \mathit{Vars} \to \widehat{\mathit{Values}}$
	other	(function definitions) $\Sigma : f \mapsto \lambda x.e$	(store) $\varsigma : \mathit{Addrs} \to \widehat{\mathit{Values}}$
Operations	focusing	$\mathbf{focus}\ (\pi, \theta)$ Fig. 3.6	$\widehat{\mathbf{focus}}\ (\widehat{\pi}, \widehat{\tau}) \in \widehat{\mathit{Types}}$ Fig. 3.12 $\widehat{\mathbf{focus}}_{\varsigma}\ (\widehat{\pi}, \widehat{u}) \in \widehat{\mathit{ValuExprs}}$ Fig. 3.16
	specialization	$\tau/p \in \mathit{Types}$ Fig. 3.3	$\widehat{\tau}/p \subsetneq \mathit{Patterns} \times \widehat{\mathit{Types}}$ Fig. 3.14
	misc.	$\sqcap : \mathit{Patterns} \times \mathit{Patterns} \to \mathit{Patterns}$ Fig. 3.4	$\mathbf{shape\_of}_{\Delta,\varsigma}(\widehat{\theta}) \in \widehat{\mathit{Patterns}}$ Fig. 3.19 $\mathbf{shatter}_{\Delta}(\widehat{\tau}) \subsetneq \widehat{\mathit{Paths}} \times \mathit{Paths} \times \widehat{\mathit{Types}}$ Fig. 3.13
Typing	Typing and Validity	$\Delta, \Gamma \vdash \theta : \tau$ Fig. 3.22 $\Delta, \Gamma \vdash e : (\tau \text{ as } \widehat{\tau})$ Fig. 3.23	$\Delta \vDash \widehat{\tau}$ Figs. 3.20 and 3.21 $\Delta, \Gamma, \varsigma \vdash \widehat{e} : \widehat{\tau}$ Fig. 3.24
	agreement	$\mathbf{agree}_{\Delta}(\tau, \widehat{\tau})$ Definition 3.2	
Semantics	pattern matching	$\sigma \vdash p \rhd u$ Fig. 3.26	$\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v}$ Fig. 3.29
	expression evaluation	$\Sigma \vdash S \hookrightarrow S'$ Fig. 3.25 $S = (\Gamma, \sigma, e)$, normal form iff. $e = (u : \tau \text{ as } \widehat{\tau})$	$\Delta, \Sigma \vdash \widehat{S} \twoheadrightarrow \widehat{S}'$ Figs. 3.27 and 3.30 $\widehat{S} = (\Gamma, \widehat{\sigma}, \varsigma, \widehat{e})$, normal form iff. $\widehat{e} \in \widehat{\mathit{Values}}$

Figure 3.34: Index of Ribbitulus notations.

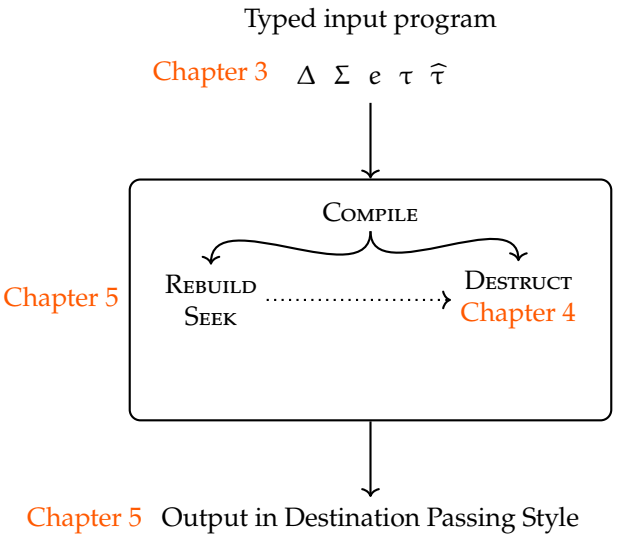
Part II

Compiling Ribbit

In the previous part of this thesis, we introduced the Ribbit language and its formalization (the Ribbitulus). We now focus on compiling this source language to low-level target code which precisely manipulates memory contents.

A central component of the Ribbitulus is the use of *memory types* to link high-level values to their desired memory representation. These custom memory layouts significantly impact the compilation of two aspects of the source language, namely pattern matching and data constructors with variable accessors. We will cover their compilation in Chapter 4 and Chapter 5 respectively.

As we will see, the compilation technique we develop in Chapter 4 for pattern matching is also a key component of our global compilation approach for the full Ribbitulus presented in Chapter 5. As such, we will wrap our pattern matching compiler in a DESTRUCT interface which will be used both by the toplevel compilation function COMPILE and by the specific procedures for data constructors REBUILD and SEEK. The following diagram gives an overview of our global compilation chain:



As shown in the previous diagram, our compilation target is a custom program representation in Destination-Passing Style, for which we also define a formal execution model. We will prove our compilation algorithms correct by showing that they emit target code whose behavior is simulated by the source program's memory-level semantics.

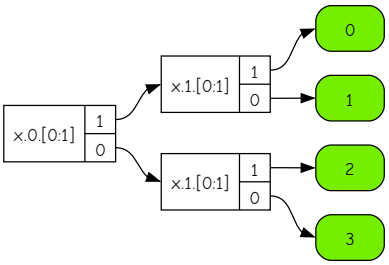
Chapter 4

Compilation of Pattern Matching

This chapter covers pattern matching compilation for the ribbit language. Our goal is to compile a list of high-level patterns to low-level code with equivalent semantics, that is, which inspects a memory value representing a given high-level value and returns the identifier of the first pattern matching this value. For instance, consider the ribbit program in Fig. 4.1a, which reuses the `Zarith` layout from Section 2.2.

```
type ZarithPair = (Zarith, Zarith);
represented as
{{(.0 as Zarith), (.1 as Zarith)}}

fn leq(x: ZarithPair) -> bool {
  match x {
    (Small(_), Small(_)) => ..., // 0
    (Large(_), Large(_)) => ..., // 1
    (Small(_), Large(_)) => ..., // 2
    (Large(_), Small(_)) => ..., // 3
  }
}
```

(a) Source code



(b) Output decision tree

Figure 4.1: Running example: `leq` comparison function on pairs of Zarith integers.

The `leq` function operates on pairs of Zarith integers represented as a two-field struct, and determines which integer in a pair is the largest depending on whether the two individual integers are both "small" (63-bit), both "large" (128-bit, stored behind a pointer) or a mixed combination. Throughout this chapter, we will use this program as a running example to illustrate the process through which we emit the low-level code depicted in Fig. 4.1b. This *decision tree* consists of leaves carrying a pattern identifier (0, 1, 2 or 3), and of decision nodes akin to C switches inspecting a given location in memory.

More generally, we aim to emit low-level code which is equivalent to a given pattern matching expression. Perhaps the simplest way to achieve this is to do a *linear* scan over each patterns, leveraging the same tools as memory-level pattern matching evaluation from Section 3.3.2.1. For each pattern, it would test whether it matches the input value, repeating the process until a match is found. The first step would be to compile each high-level pattern to an equivalent list of *memory patterns* using **pat2mem**. We could then emit low-level code acting as an interpreter for the memory pattern matching judgment $\blacktriangleright$ for this particular list of memory patterns. While correct (as proven in Theorem 3.2), this naive approach is very inefficient, as demonstrated by the following example.

Example 4.1 (Naive approach for `leq`)**.** We model `Zarith` and `ZarithPair`, along with their memory layouts, as the following high-level and memory types (using a 128-bit wide integer to emulate

`GMP::BigInt`):

$$\tau_{\text{zarith}} = \text{Small}(\text{I}_{63}) \mid \text{Large}(\text{I}_{128}) \qquad \tau_{\text{2zarith}} = \langle \tau_{\text{zarith}}, \tau_{\text{zarith}} \rangle$$

$$\widehat{\tau}_{\text{zarith}} = \text{split}\,(.[0:1]) \left\{ \begin{array}{lll} 1 & \text{from} & \text{Small}(\_) \Rightarrow \_{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : (.\text{Small as } \text{I}_{63}) \\ 0 & \text{from} & \text{Large}(\_) \Rightarrow \&_{64}\,((.\text{Large as } \text{I}_{128})) \ltimes [0:1] : (0)_1 \end{array} \right\}$$

$$\widehat{\tau}_{\text{2zarith}} = \{\!\{(.0 \text{ as } \widehat{\tau}_{\text{zarith}}), (.1 \text{ as } \widehat{\tau}_{\text{zarith}})\}\!\}$$

The $\widehat{\tau}_{\text{zarith}}$ memory type encodes small integers directly in a 64-bit word whose lowest bit is set to 1 to distinguish it from pointers, and boxes large integers into a 64-bit pointer whose lowest bit is set to 0. As specified in the source program Fig. 4.1a, $\widehat{\tau}_{\text{2zarith}}$ encodes a pair of Zarith integers as a two-field struct.

The `leq` pattern matching function corresponds to the four following high-level patterns of type τ_{2zarith}, associated with their respective identifiers 0, 1, 2 and 3:

$$\text{p}_0 = \langle \text{Small}(\_), \text{Small}(\_) \rangle \qquad \text{p}_1 = \langle \text{Small}(\_), \text{Large}(\_) \rangle$$
$$\text{p}_2 = \langle \text{Large}(\_), \text{Small}(\_) \rangle \qquad \text{p}_3 = \langle \text{Large}(\_), \text{Large}(\_) \rangle$$

Using **pat2mem**, we get the four following memory patterns, each representing one high-level pattern according to $\widehat{\tau}_{\text{2zarith}}$:

$$\widehat{\text{p}}_0 = \{\!\{\_{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : \_{63}, \_{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : \_{63}\}\!\}$$
$$\widehat{\text{p}}_1 = \{\!\{\&_{64}\,(\_{128}) \ltimes [0:1] : (0)_1, \_{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : \_{63}\}\!\}$$
$$\widehat{\text{p}}_2 = \{\!\{\_{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : \_{63}, \&_{64}\,(\_{128}) \ltimes [0:1] : (0)_1\}\!\}$$
$$\widehat{\text{p}}_3 = \{\!\{\&_{64}\,(\_{128}) \ltimes [0:1] : (0)_1, \&_{64}\,(\_{128}) \ltimes [0:1] : (0)_1\}\!\}$$

In order to match a memory value of type $\widehat{\tau}_{\text{2zarith}}$ against each of these four memory patterns, we must inspect the tag – that is, the least significant bit, which distinguishes between Small and Large constructors – of both fields. This potentially adds up to $4 \times 2 = 8$ computations, as demonstrated by the following pseudo-code:

```
if (x.0 & 1 == 1 && x.1 & 1 == 1)
    return 0;
else if (x.0 & 1 == 0 && x.1 & 1 == 1)
    return 1;
else if (x.0 & 1 == 1 && x.1 & 1 == 0)
    return 2;
else if (x.0 & 1 == 0 && x.1 & 1 == 0)
    return 3;
else
    return -1;
```

This is considerably less efficient than the decision tree shown in Fig. 4.1b, which achieves the same semantics with only three switch nodes in total, and only two in each possible code path. △

4.1 Problem statement

Following the TMatch typing rule from Fig. 3.23, we model a well-typed pattern matching expression of type τ in the type variable environment Δ as a list of N patterns $\{\text{p}_0, \ldots, \text{p}_{\text{N}-1}\}$ such that each pattern p_j is of type τ and every value of type τ is matched by at least one of these patterns. Given a memory type $\widehat{\tau}$ in agreement with τ, our goal is to emit executable code which, given the memory representation according to $\widehat{\tau}$ of a value v of type τ as input, returns the smallest identifier $\text{j} \in \{0, \ldots, \text{N}-1\}$ such that p_j matches v.

In our pattern matching compilation approach, we first lower each high-level pattern p_j to a list of memory patterns using $\textbf{pat2mem}_\Delta(\text{p}_\text{j}, \widehat{\tau})$. As Example 4.1 shows, matching against each memory pattern sequentially yields highly redundant code. A more efficient strategy is to emit a *decision tree* such

as the one shown in Fig. 4.1b. Each of its switch nodes inspects a given location in memory indicated by a memory path $\widehat{\pi}$, while each leaf indicates the identifier $j \in \mathbb{N}$ of the first high-level pattern which matches the input value. Previous works on pattern matching compilation provide approaches emitting compact and efficient decision trees, but do not handle custom memory layouts.

Our approach relies on a bespoke intermediate representation dubbed *memory trees* to compile memory patterns to decision trees. We assume patterns are exhaustive and non-redundant. This is enforced by our typing judgment and can be achieved with well-known techniques (Maranget 2007). We define memory trees in Section 4.2, then detail our compilation algorithms in Section 4.3 and prove them correct in Section 4.4. Section 4.5 covers related work on pattern matching compilation.

4.2 Intermediate Representation: Memory Trees

Memory trees are a superset of decision trees. In addition to switches and leaves, they include constructs which allow us to encode fine notions of *dependency* which arise from splits in memory types. As we have seen in Section 3.1, *splits* are an essential part of the Ribbitulus, allowing us to handle case disjunction gracefully, even in cases where the discriminant between branches is found in unexpected places. Therefore, we must enforce throughout our compilation process that we respect split dependencies, i.e., we always inspect a split's *discriminant location*, which distinguishes between different constructors, *before* accessing its branches' contents. For instance, we must check that a memory value is indeed a pointer by inspecting its tag before dereferencing it.

4.2.1 Syntax

Memory trees, denoted $\mathcal{T}$ and defined in Fig. 4.2, are our main intermediate representation during pattern matching compilation. The key idea is to preserve dependencies, yet leave the compiler free to arrange independent operations in any order. Similar to decision trees, a memory tree can be a leaf (J) where J is the list of output branch identifiers that accept memory values for which evaluation reaches this point, or a "decision node" $\mathsf{switch}(\widehat{\pi})\{\dots\}$ which inspects the position $\widehat{\pi}$ in memory and picks a branch accordingly. Each branch consists of an immediate c on its left-hand side and of a memory tree on its right-hand side. As a special case, the last branch may be a default branch denoted $\_ \to \mathcal{T}$, which catches any memory value whose subterm at position $\widehat{\pi}$ does not match any previous branch. In the rest of this chapter, switch nodes with a gray default branch appearing in a definition or statement indicate that it applies to both kinds of switches (with and without a default branch). A tree can also be a *bud* (J as $\widehat{\tau}$): a leaf which already carries a set of accepting output identifiers J, but could still be developed further if needed using the memory type $\widehat{\tau}$. Finally, trees can be assembled "in parallel": $\mathcal{T} \parallel \mathcal{T}'$ is a tree where one of $\mathcal{T}$ and $\mathcal{T}'$ is executed first, and the other second, the order being not yet decided.

$$
\begin{aligned}
J &::= \{j_0, \dots, j_{n-1}\} && \text{(list of case identifiers)} \\
\mathcal{T} &::= (J) && \text{(leaf with list } J \text{ of possible outputs)} \\
&\mid (J \text{ as } \widehat{\tau}) && \text{(fragment bud with subterm layout } \widehat{\tau} \text{ and list } J \text{ of possible outputs)} \\
&\mid \mathcal{T}_0 \parallel \dots \parallel \mathcal{T}_{n-1} && \text{(parallel node with } n \text{ branches)} \\
&\mid \mathsf{switch}(\widehat{\pi})\,\{c_0 \to \mathcal{T}_0, \dots, c_{n-1} \to \mathcal{T}_{n-1}, \_ \to \mathcal{T}'\} \\
& && \text{(switch node with } n \text{ cases and an optional default branch)}
\end{aligned}
$$

Figure 4.2: Memory trees.

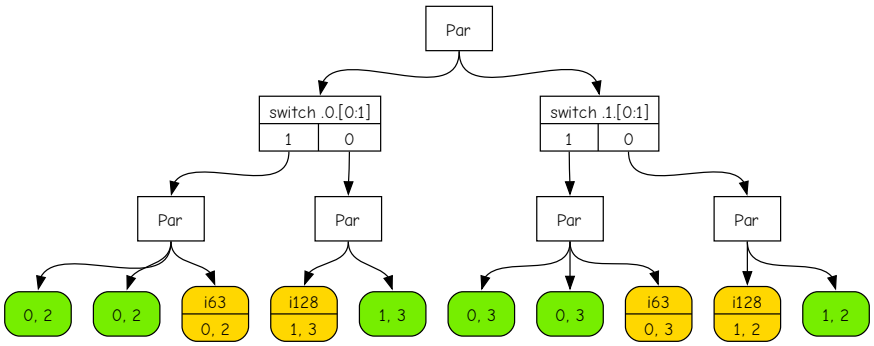
Example 4.2 (Intermediate memory tree from Zarith leq compilation)**.** The memory tree $\mathcal{T} = \mathcal{T}_l \parallel \mathcal{T}_r$

appears during the compilation of the `leq` function for the $\widehat{\tau}_{2\text{zarith}}$ layout from Fig. 4.1a, where:

$$\mathcal{T}_l = \text{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 1 & \to & (\{0,2,3\}) \,\|\, (\{0,2,3\}) \,\|\, (\{0,2,3\} \text{ as } I_{63}) \\ 0 & \to & (\{1,3\} \text{ as } I_{128}) \,\|\, (\{1,3\}) \end{array} \right\}$$

$$\mathcal{T}_r = \text{switch}(.1.[0:1]) \left\{ \begin{array}{lll} 1 & \to & (\{0,2,3\}) \,\|\, (\{0,2,3\}) \,\|\, (\{0,2,3\} \text{ as } I_{63}) \\ 0 & \to & (\{1,2\} \text{ as } I_{128}) \,\|\, (\{1,2\}) \end{array} \right\}$$

Its graphical representation is:



For $\text{switch}(\widehat{\pi}) \left\{ c_i \to \mathcal{T}_i \;\middle|\; 1 \leqslant i \leqslant n \right\}$, the memory path $\widehat{\pi}$ is at the top of the node and each concrete value c_i labels a branch to its subtree $\mathcal{T}_i$. Buds are depicted in yellow, leaves in green. Empty sets are not displayed. △

As a shortcut, we define a mapping operation denoted $\mathcal{T}\,[f]$ in Fig. 4.3, which substitutes the output identifier set J in each leaf and bud of the memory tree $\mathcal{T}$ with $f(J)$. For instance, $\mathcal{T}\,[(\{j_1,\ldots,j_n\}) \mapsto (\{j_1,\ldots,j_n,j\})]$ adds the output identifier j to each leaf in $\mathcal{T}$.

$$(J)\,[f] = (f(J)) \qquad (J \text{ as } \widehat{\tau})\,[f] = (f(J) \text{ as } \widehat{\tau}) \qquad \mathcal{T}_1 \,\|\ldots\|\, \mathcal{T}_n\,[f] = \mathcal{T}_1\,[f] \,\|\ldots\|\, \mathcal{T}_n\,[f]$$

$$\text{switch}(\widehat{\pi}) \left\{ \begin{array}{lll} c_1 & \to & \mathcal{T}_1 \\ \ldots & \to & \ldots \\ c_n & \to & \mathcal{T}_n \\ \_ & \to & \mathcal{T}' \end{array} \right\} [f] = \text{switch}(\widehat{\pi}) \left\{ \begin{array}{lll} c_1 & \to & \mathcal{T}_1\,[f] \\ \ldots & \to & \ldots \\ c_n & \to & \mathcal{T}_n\,[f] \\ \_ & \to & \mathcal{T}'\,[f] \end{array} \right\}$$

Figure 4.3: Mapping operation on memory trees.

4.2.2 Semantics

In this section, we define the big-step evaluation of a memory tree $\mathcal{T}$ on an input consisting of a memory value $\widehat{v}$ and a store ς. This judgment, denoted $\varsigma, \widehat{v} \vdash \mathcal{T} \blacktriangleright\!\blacktriangleright J$ and defined in Fig. 4.4, returns the ordered set J of identifiers which correspond to patterns matching $\widehat{v}$. For leaves and buds, we simply return the pattern identifier set J that they carry. In a "Par" node, each subtree corresponds to a different part of the memory value being inspected; in order to match a specific memory pattern, each of these parts must match its associated sub-pattern. That is, the set of patterns which match the whole memory value is the *intersection* of the sets of patterns which match each of its parts. Finally, we evaluate a switch on the memory location $\widehat{\pi}$ by focusing on this position in the memory value being inspected, which must contain a constant value at this position (ensured by typing); depending on this value, we either take the case branch associated with this constant (ESwitchCase) or the default branch (ESwitchDef).

$$\textsc{ELeaf}\ \frac{}{\varsigma,\widehat{v}\vdash (\mathrm{J}) \blacktriangleright\!\blacktriangleright \mathrm{J}} \qquad \textsc{EBud}\ \frac{}{\varsigma,\widehat{v}\vdash (\mathrm{J}\ \text{as}\ \widehat{\tau}) \blacktriangleright\!\blacktriangleright \mathrm{J}}$$

$$\textsc{ESwitchCase}\ \frac{\widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi},\widehat{v}) = (c_i)_\ell \qquad \varsigma,\widehat{v}\vdash \mathcal{T}_i \blacktriangleright\!\blacktriangleright \mathrm{J}}{\varsigma,\widehat{v}\vdash \mathsf{switch}(\widehat{\pi})\{c_0\to\mathcal{T}_0,\ldots,c_{n-1}\to\mathcal{T}_{n-1},\_\to\mathcal{T}'\} \blacktriangleright\!\blacktriangleright \mathrm{J}}$$

$$\textsc{EPar}\ \frac{\varsigma,\widehat{v}\vdash \mathcal{T}_i \blacktriangleright\!\blacktriangleright \mathrm{J}_i}{\varsigma,\widehat{v}\vdash \mathcal{T}_0\,\|\ldots\|\,\mathcal{T}_{n-1} \blacktriangleright\!\blacktriangleright \bigcap_{0\leqslant i<n} \mathrm{J}_i}$$

$$\textsc{ESwitchDef}\ \frac{\widehat{\mathbf{focus}}_\varsigma\,(\widehat{\pi},\widehat{v}) = (c)_\ell \qquad c\notin\{c_0,\ldots,c_{n-1}\} \qquad \varsigma,\widehat{v}\vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright \mathrm{J}}{\varsigma,\widehat{v}\vdash \mathsf{switch}(\widehat{\pi})\{c_0\to\mathcal{T}_0,\ldots,c_{n-1}\to\mathcal{T}_{n-1},\_\to\mathcal{T}'\} \blacktriangleright\!\blacktriangleright \mathrm{J}}$$

Figure 4.4: Memory tree evaluation.

Example 4.3 (Evaluation of a memory tree for our running example.)**.** Let $\mathcal{T} = \mathcal{T}_l \,\|\, \mathcal{T}_r$ the memory tree described in Example 4.2. Consider the following memory value and store, which represent the high-level value $\langle \mathrm{Small}(42), \mathrm{Large}(7)\rangle$ according to $\widehat{\tau}_{2\mathrm{zarith}}$:

$$\widehat{v} = \{\!\{ \__{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : (42)_{63}, \&_{64}(a) \ltimes [0:1] : (0)_1 \}\!\}$$
$$\varsigma = \{a \mapsto (7)_{128}\}$$

We evaluate this tree on the input $\varsigma, \widehat{v}$, starting with $\mathcal{T}_l$:

$$\dfrac{\widehat{\mathbf{focus}}_\varsigma\,(.0.[0:1],\widehat{v}) = (1)_1 \qquad \dfrac{\dfrac{}{\varsigma,\widehat{v}\vdash(\{0,2,3\}) \blacktriangleright\!\blacktriangleright \{0,2,3\}}\textsc{ELeaf} \qquad \dfrac{}{\varsigma,\widehat{v}\vdash(\{0,2,3\}\ \text{as}\ \mathrm{I}_{63}) \blacktriangleright\!\blacktriangleright \{0,2,3\}}\textsc{EBud}}{\varsigma,\widehat{v}\vdash(\{0,2,3\})\,\|\,(\{0,2,3\})\,\|\,(\{0,2,3\}\ \text{as}\ \mathrm{I}_{63}) \blacktriangleright\!\blacktriangleright \{0,2,3\}}\textsc{EPar}}{\varsigma,\widehat{v}\vdash \mathcal{T}_l \blacktriangleright\!\blacktriangleright \{0,2,3\}}\textsc{ESwitchCase}$$

then $\mathcal{T}_r$:

$$\dfrac{\widehat{\mathbf{focus}}_\varsigma\,(.1.[0:1],\widehat{v}) = (0)_1 \qquad \dfrac{\dfrac{}{\varsigma,\widehat{v}\vdash(\{1,2\}\ \text{as}\ \mathrm{I}_{128}) \blacktriangleright\!\blacktriangleright \{1,2\}}\textsc{EBud} \qquad \dfrac{}{\varsigma,\widehat{v}\vdash(\{1,2\}) \blacktriangleright\!\blacktriangleright \{1,2\}}\textsc{ELeaf}}{\varsigma,\widehat{v}\vdash(\{1,2\}\ \text{as}\ \mathrm{I}_{128})\,\|\,(\{1,2\}) \blacktriangleright\!\blacktriangleright \{1,2\}}\textsc{EPar}}{\varsigma,\widehat{v}\vdash \mathcal{T}_r \blacktriangleright\!\blacktriangleright \{1,2\}}\textsc{ESwitchCase}$$

and finally $\mathcal{T}$, which yields the intersection of the two previous results:

$$\textsc{EPar}\ \frac{\varsigma,\widehat{v}\vdash \mathcal{T}_l \blacktriangleright\!\blacktriangleright \{0,2,3\} \qquad \varsigma,\widehat{v}\vdash \mathcal{T}_r \blacktriangleright\!\blacktriangleright \{1,2\}}{\varsigma,\widehat{v}\vdash \mathcal{T}_l\,\|\,\mathcal{T}_r \blacktriangleright\!\blacktriangleright \{2\}}$$

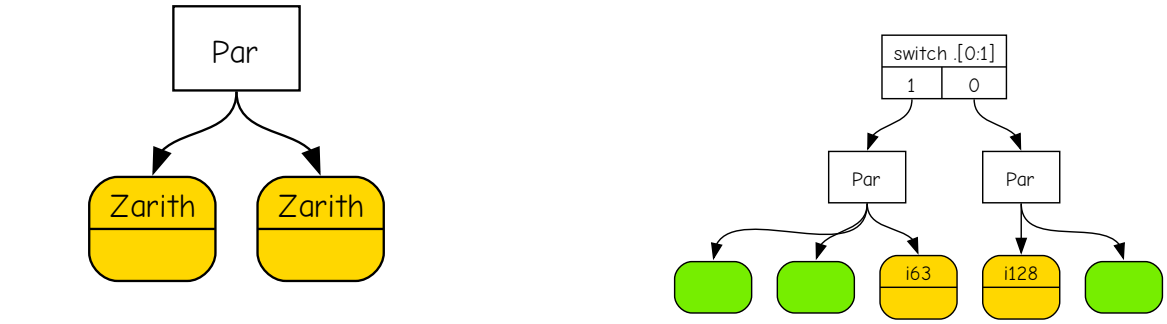
The final result is the singleton $\{2\}$, which expresses that only the third pattern ($\langle \mathrm{Small}(\_), \mathrm{Large}(\_)\rangle$) from the initial pattern matching matches the value ($\langle \mathrm{Small}(42), \mathrm{Large}(7)\rangle$) represented by $\widehat{v}$. △

4.3 From Memory Patterns To Memory Trees

We now describe our actual compilation procedure. Given a memory type $\widehat{\tau}$, we now consider a list of branches $\{\widehat{p}_i \to j_i \mid 0 \leqslant i < n\}$, each consisting of a memory pattern $\widehat{p}_i$ of type $\widehat{\tau}$ (obtained via **pat2mem**) and its *output identifier* j_i. In order to compile it to an equivalent sequential decision tree – that is, whose evaluation on a memory value of type $\widehat{\tau}$ yields the identifiers of all matching memory patterns – we proceed in three steps:

1. Scaffold (Section 4.3.1) a memory tree template from the memory type $\widehat{\tau}$.
2. Weave (Section 4.3.2) each memory pattern onto the current tree.
3. Finalize (Section 4.3.3) the memory tree by sequentializing and optimizing it.

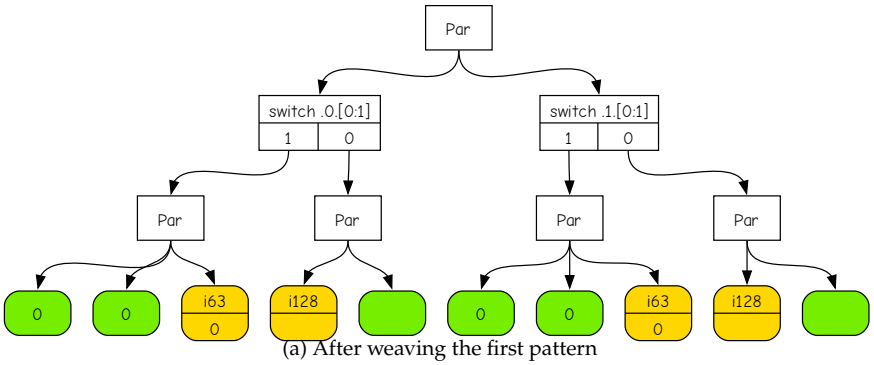
The detailed compilation process will be illustrated on our running example. The successive memory trees are depicted graphically in Figs. 4.5 to 4.7.



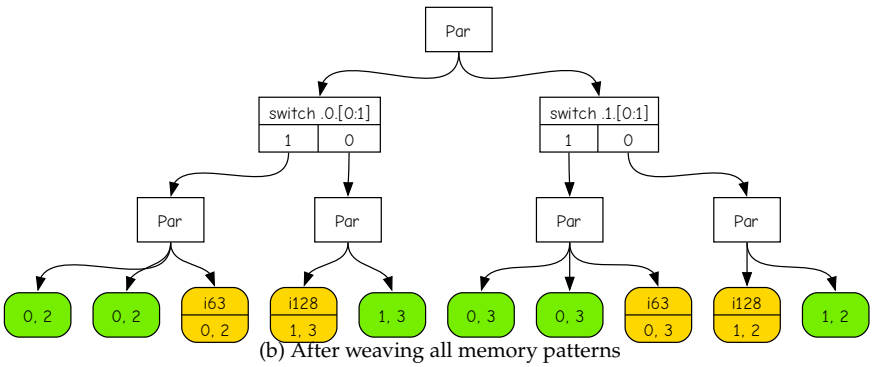
(a) After scaffolding $\widehat{\tau}_{\text{2zarith}}$ (first tree for the running example)

(b) After scaffolding $\widehat{\tau}_{\text{zarith}}$ (grown during weaving)

Figure 4.5: First compilation step: scaffolding.



(a) After weaving the first pattern



(b) After weaving all memory patterns

Figure 4.6: Second compilation step for the running example: weaving.

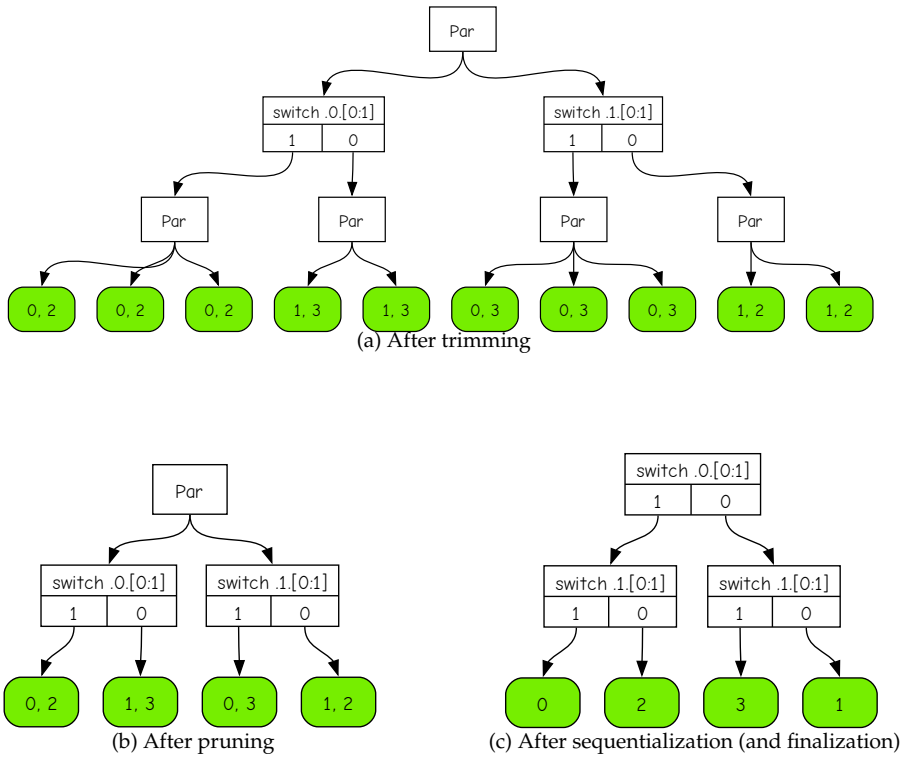


Figure 4.7: Last compilation steps for the running example.

4.3.1 Scaffolding: Memory Type-Specific Tree Templates

Scaffolding builds a memory tree "template" based on a memory type. This memory tree does not yet contain any actual output branch, since no pattern has been taken into account yet. More precisely, $\textsc{Scaffold}_\Delta(\mathrm{J}, \widehat{\pi}, \widehat{\tau})$, defined in Fig. 4.8, creates a memory tree based on the position $\widehat{\pi}$ (initialized as ε) in the type $\widehat{\tau}$, with J placed at the leaves. Constant and empty word types are directly turned into leaves, as they do not involve any choice. Fragments (π as $\widehat{\tau}$) are turned into buds (J as $\widehat{\tau}$), keeping the type $\widehat{\tau}$ available for later expansion by nested patterns. Struct and composite types are all treated as parallel nodes: indeed, the order in which to explore fields is irrelevant, and will be determined later on. Splits are turned into switch nodes that inspect their discriminant position; splits with multiple discriminants become a parallel node containing a switch for each discriminant. Note that, unlike splits, the discriminant of a switch is absolute, hence the use of $\widehat{\pi}.\widehat{\pi}'$. Additionally, provenances are not useful at this stage anymore, and are thus not recorded in the memory tree. Finally, primitive types I_ℓ are represented by a switch on the current position with an initial default branch accepting all values, which will be used as a simple C-like switch on integers.

$$
\begin{array}{l}
\textsc{Scaffold}_\Delta(J, \widehat{\pi})\{ \\
\quad t \longrightarrow \textsc{Scaffold}_\Delta(J, \widehat{\pi}, \Delta(t)) \\
\quad (c)_\ell \longrightarrow (J) \\
\quad \__\ell \longrightarrow (J) \\
\quad I_\ell \longrightarrow \mathsf{switch}(\widehat{\pi})\{\_ \to (J)\} \\
\quad \&_\ell\,(\widehat{\tau}) \longrightarrow \textsc{Scaffold}_\Delta(J, \widehat{\pi}.*, \widehat{\tau}) \\
\quad \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i \longrightarrow \textsc{Scaffold}_\Delta(J, \widehat{\pi}.\neg r_0 \ldots \neg r_{n-1}, \widehat{\tau}) \,\|_{0 \leqslant i < n}\, \textsc{Scaffold}_\Delta(J, \widehat{\pi}.r_i, \widehat{\tau}_i) \\
\quad \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\} \longrightarrow \textsc{Scaffold}_\Delta(J, \widehat{\pi}.0, \widehat{\tau}_0) \,\|\, \ldots \,\|\, \textsc{Scaffold}_\Delta(J, \widehat{\pi}.(n-1), \widehat{\tau}_{n-1}) \\
\quad (\pi \text{ as } \widehat{\tau}) \longrightarrow (J \text{ as } \widehat{\tau}) \\
\quad \mathsf{split}\,(\widehat{\pi}_1, \ldots, \widehat{\pi}_N)\,\{ \\
\qquad c_{1,1}, \ldots, c_{1,N} \text{ from } P_1 \Rightarrow \widehat{\tau}_1 \\
\qquad \ldots \qquad\quad \text{from } \ldots \Rightarrow \ldots \longrightarrow \|_{1 \leqslant i \leqslant N} \left(\begin{array}{l} \mathsf{switch}(\widehat{\pi}.\widehat{\pi}_i)\{ \\ \quad c_{1,i} \to \textsc{Scaffold}_\Delta(J, \widehat{\pi}, \widehat{\tau}_1) \\ \quad \ldots \to \ldots \\ \quad c_{n,i} \to \textsc{Scaffold}_\Delta(J, \widehat{\pi}, \widehat{\tau}_n) \\ \} \end{array} \right) \\
\qquad c_{n,1}, \ldots, c_{n,N} \text{ from } P_n \Rightarrow \widehat{\tau}_n \\
\quad \} \\
\}
\end{array}
$$

Figure 4.8: Scaffold a memory tree from a memory type, a list of accepted identifiers J and a base memory location $\widehat{\pi}$.

Example 4.4 (Scaffolded tree from $\widehat{\tau}_{2\text{zarith}}$)**.** Recall the memory type $\widehat{\tau}_{2\text{zarith}} = \{\!\{(.0 \text{ as } \widehat{\tau}_{\text{zarith}}), (.1 \text{ as } \widehat{\tau}_{\text{zarith}})\}\!\}$ from Fig. 4.1a. We begin with $\textsc{Scaffold}(\varnothing, \varepsilon, \widehat{\tau}_{2\text{zarith}})$. The resulting memory tree is the following, as depicted in Fig. 4.5a:

$$\mathcal{T}_{-1} = (\varnothing \text{ as } \widehat{\tau}_{\text{zarith}}) \,\|\, (\varnothing \text{ as } \widehat{\tau}_{\text{zarith}})$$

The "struct" rule generates a parallel node; its two children are generated from the fragments $(.0 \text{ as } \widehat{\tau}_{\text{zarith}})$ and $(.1 \text{ as } \widehat{\tau}_{\text{zarith}})$, which both yield a bud of type $\widehat{\tau}_{\text{zarith}}$. △

4.3.2 Weaving Patterns Into A Memory Tree

We can now *weave* each memory pattern onto the previously generated memory tree. For each memory pattern branch $(\widehat{p}_i \to j_i)$, we define $\mathcal{T}_i = \textsc{Weave}_\Delta(j_i, \varepsilon, \widehat{\tau}, \widehat{p}_i, \mathcal{T}_i)$ until exhaustion of all patterns. The initial tree $\mathcal{T}_{-1}$ is the output of the scaffolding phase. Each weaving adds relevant choices and outputs from the pattern $\widehat{p}_i$ to the tree. The general form $\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T})$, defined in Fig. 4.9, takes a memory pattern $\widehat{p}$ and a tree $\mathcal{T}$, along with the current path $\widehat{\pi}$, memory layout $\widehat{\tau}$ and an output identifier j, and returns a new memory tree. It inspects both pattern and tree, and integrates the latter into the former.

Branch identifiers, leaves and wildcards The general goal of weaving is to add the branch identifier j to each leaf or bud that is relevant to this pattern. By design of scaffolding, leaves always correspond to memory types for which no more inspection is necessary (i.e., constant or empty word types); the WeaveLeaf rule simply adds the output identifier j to the list of accepting branches. Conversely, wildcard patterns accept all inputs and the WeaveWildcard rule simply adds a wildcard's output identifier to every leaf and bud in the tree.

Fragments, buds and tree expansion Fragments and buds enable the memory tree to be expanded as needed to handle nested patterns. This expansion is only necessary for non-wildcard memory patterns and handled by the WeaveBud rule, which uses Scaffold to grow a new memory tree in place, then weaves the subpattern onto this new tree.

Aggregates and parallel nodes The purpose of parallel nodes is to model "aggregate" constructions, which include structs, composite words and (technically) pointers. In all these cases, the order in which sub-patterns should be explored is not set in stone, and will be decided during sequentialization (Section 4.3.3) based on heuristics. The WeavePointer rule simply weaves the memory pattern below the pointer, since a pointer only "aggregates" one field (its pointee). WeaveComposite and WeaveStruct recursively explore every subtree, each corresponding to a struct field or composite word element. Note how we rely on the fact that the order of the subtrees in unchanged during the weaving phase.

WeaveWildcard

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \__\ell, \mathcal{T}) = \mathcal{T} \left[\begin{array}{ll} (J) & \mapsto (J \cup \{j\}) \\ (J \text{ as } \widehat{\tau}) & \mapsto (J \cup \{j\} \text{ as } \widehat{\tau}) \end{array} \right]$$

WeaveLeaf

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, (J)) = (J \cup \{j\})$$

WeaveTyVar

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, t, \widehat{p}, \mathcal{T}) = \textsc{Weave}_\Delta(j, \widehat{\pi}, \Delta(t), \widehat{p}, \mathcal{T})$$

WeaveFragment

$$\frac{\mathcal{T} \text{ is not a bud}}{\textsc{Weave}_\Delta(j, \widehat{\pi}, (\pi \text{ as } \widehat{\tau}), \widehat{p}, \mathcal{T}) = \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T})}$$

WeaveBud

$$\frac{\widehat{p} \text{ is not a wildcard pattern}}{\textsc{Weave}_\Delta(j, \widehat{\pi}, (\pi \text{ as } \widehat{\tau}), \widehat{p}, (J \text{ as } \widehat{\tau})) = \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \textsc{Scaffold}_\Delta(J, \widehat{\pi}, \widehat{\tau}))}$$

WeavePrimitiveCase

$$\textsc{Weave}_\Delta\left(j, \widehat{\pi}, I_\ell, (c_i)_\ell, \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{ll} c_0 & \to (J_0) \\ \dots & \to \dots \\ c_i & \to (J_i) \\ \dots & \to \dots \\ c_{n-1} & \to (J_{n-1}) \\ \_ & \to (J) \end{array} \right\}\right) = \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{ll} c_0 & \to (J_0) \\ \dots & \to \dots \\ c_i & \to (J_i \cup \{j\}) \\ \dots & \to \dots \\ c_{n-1} & \to (J_{n-1}) \\ \_ & \to (J) \end{array} \right\}$$

WeavePrimitiveDefault

$$\frac{c \notin \{c_0, \dots, c_{n-1}\}}{\textsc{Weave}_\Delta\left(j, \widehat{\pi}, I_\ell, (c)_\ell, \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{ll} c_0 & \to (J_0) \\ \dots & \to \dots \\ c_{n-1} & \to (J_{n-1}) \\ \_ & \to (J) \end{array} \right\}\right) = \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{ll} c_0 & \to (J_0) \\ \dots & \to \dots \\ c_{n-1} & \to (J_{n-1}) \\ c & \to (J \cup \{j\}) \\ \_ & \to (J) \end{array} \right\}}$$

WeavePointer

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \&_\ell(\widehat{\tau}), \&_\ell(\widehat{p}), \mathcal{T}) = \textsc{Weave}_\Delta(j, \widehat{\pi}.*, \widehat{\tau}, \widehat{p}, \mathcal{T})$$

WeaveComposite

$$\frac{\mathcal{T}' = \textsc{Weave}_\Delta(j, \widehat{\pi}.\neg r_0 \dots \neg r_{n-1}, \widehat{\tau}, \widehat{p}, \mathcal{T}) \qquad \mathcal{T}_i' = \textsc{Weave}_\Delta(j, \widehat{\pi}.r_i, \widehat{\tau}_i, \widehat{p}_i, \mathcal{T}_i)}{\textsc{Weave}_\Delta\left(j, \widehat{\pi}, \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i, \widehat{p} \bowtie_{0 \leqslant i < n} r_i : \widehat{p}_i, \mathcal{T} \parallel_{0 \leqslant i < n} \mathcal{T}_i\right) = \mathcal{T}' \parallel_{0 \leqslant i < n} \mathcal{T}_i'}$$

WeaveStruct

$$\frac{\mathcal{T}_i' = \textsc{Weave}_\Delta(j, \widehat{\pi}.i, \widehat{\tau}_i, \widehat{p}_i, \mathcal{T}_i)}{\textsc{Weave}_\Delta(j, \widehat{\pi}, \{\!\{\widehat{\tau}_0, \dots, \widehat{\tau}_{n-1}\}\!\}, \{\!\{\widehat{p}_0, \dots, \widehat{p}_{n-1}\}\!\}, \mathcal{T}_0 \| \dots \| \mathcal{T}_{n-1}) = \mathcal{T}_0' \| \dots \| \mathcal{T}_{n-1}'}$$

WeaveSplit

$$\frac{\begin{array}{c} \widehat{\tau} = \mathrm{split}(\widehat{\pi}_0, \dots, \widehat{\pi}_{N-1}) \left\{ c_{i,0}, \dots, c_{i,N-1} \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n \right\} \\ \mathcal{T}_{sk} = \mathsf{switch}(\widehat{\pi}.\widehat{\pi}_k) \left\{ c_{i,k} \to \mathcal{T}_i \;\middle|\; 0 \leqslant i < n \right\} \qquad \mathcal{T}_{sk}' = \mathsf{switch}(\widehat{\pi}.\widehat{\pi}_k) \left\{ c_{i,k} \to \mathcal{T}_i' \;\middle|\; 0 \leqslant i < n \right\} \\ \mathcal{T}_i' = \begin{cases} \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}_i, \widehat{p}, \mathcal{T}_i) & \text{if } \forall k \in \{0, \dots, N-1\}, c_{i,k} = \widehat{\textbf{focus}}(\widehat{\pi}_k, \widehat{p}) \\ \mathcal{T}_i & \text{otherwise} \end{cases} \end{array}}{\textsc{Weave}_\Delta\left(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \parallel_{0 \leqslant k < N} \mathcal{T}_{sk}\right) = \parallel_{0 \leqslant k < N} \mathcal{T}_{sk}'}$$

Figure 4.9: Weave a memory pattern onto a memory tree.

Splits, primitive types and switches Switches are decision nodes used to model sum-like constructs (i.e., splits), as well as a traditional C-like switch on primitive types. For split types, switch nodes correspond to discriminant locations, which are combined with a "Par" node. The WeaveSplit rule inspects all of a split's discriminant values simultaneously, so as to only propagate patterns to trees corresponding to split branches which are correct types for this pattern.

Primitive types are modelled as a single switch node with a default branch, which captures all possible primitive values. The memory pattern found at the position inspected by the switch should be a constant or empty word pattern. Depending on this subpattern and on existing switch branches, we use one of three possible rules. Wildcard patterns accept all values and therefore propagate to every switch branch, including the default branch if it exists, using the WeaveWildcard rule. Constant patterns only accept a specific primitive value, and will only propagate to a single switch branch. If this constant value is already present as the left-hand-side of a switch branch, we weave the pattern onto this particular branch with WeavePrimitiveCase. Otherwise, we must add a new case branch to the switch with WeavePrimitiveDefault. We use the default switch branch as a base subtree on which to weave the memory pattern; the design of Scaffold, together with typing of memory patterns, ensures that the default branch always exists in this situation.

Example 4.5 (Woven tree)**.** We can now weave the four memory patterns from Example 4.1 onto the scaffolded tree $\mathcal{T}_{-1}$ from Example 4.4. Let us first weave the memory pattern associated with the first branch (corresponding to the high-level pattern $\langle \text{Small}(\_), \text{Small}(\_) \rangle$ and to the output identifier 0): $\widehat{\mathsf{p}}_0 = \{\!\{\widehat{\mathsf{p}}_{\text{Small}}, \widehat{\mathsf{p}}_{\text{Small}}\}\!\}$ where $\widehat{\mathsf{p}}_{\text{Small}} = \__{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : \__{63}$.

The WeaveStruct rule explores the 'Par' node, whose two children correspond to the two fields of the struct. Let us compute its first child with $\textsc{Weave}(0, .0, \widehat{\tau}_{\text{2zarith}}, \widehat{\mathsf{p}}_{\text{Small}}, (\varnothing \text{ as } \widehat{\tau}_{\text{zarith}}))$. Since $\widehat{\mathsf{p}}_{\text{Small}}$ is not a wildcard memory pattern, we expand the left bud with WeaveBud, replacing it with the following memory tree (depicted in Fig. 4.5b):

$$\begin{aligned}\textsc{Scaffold}(\varnothing, .0, \widehat{\tau}_{\text{zarith}}) &= \mathsf{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 0 & \to & \textsc{Scaffold}(\varnothing, .0, \&_{64}\left((.\text{Large as } \mathrm{I}_{128})\right) \ltimes [0:1] : (0)_1) \\ 1 & \to & \textsc{Scaffold}(\varnothing, .0, \__{64} \ltimes [0:1] : (1)_1 \ltimes [1:63] : (.\text{Small as } \mathrm{I}_{63})) \end{array} \right\} \\ &= \mathsf{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 0 & \to & (\varnothing \text{ as } \mathrm{I}_{128}) \parallel (\varnothing) \\ 1 & \to & (\varnothing) \parallel (\varnothing) \parallel (\varnothing \text{ as } \mathrm{I}_{63}) \end{array} \right\}\end{aligned}$$

The split in $\widehat{\tau}_z$ is mirrored by a new switch node, on which we then weave the pattern $\widehat{\mathsf{p}}_{\text{Small}}$. We inspect the subpattern at the position inspected by the switch (removing the leading .0 since it is the position of this subpattern): $\widehat{\textbf{focus}}\,(.[0:1], \widehat{\mathsf{p}}_{\text{Small}}) = (1)_1$. Using the WeaveSplit rule, weaving will only explore the branch corresponding to the value 1. We then propagate the new identifier 0 to the bud and two leaves of this subtree, using WeaveComposite, WeaveLeaf and WeaveWildcard rules and yielding the following tree:

$$\mathsf{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 0 & \to & (\varnothing \text{ as } \mathrm{I}_{128}) \parallel (\varnothing) \\ 1 & \to & (\{0\}) \parallel (\{0\}) \parallel (\{0\} \text{ as } \mathrm{I}_{63}) \end{array} \right\}$$

We repeat the same process for the second field, which is identical, and finally get the result of $\textsc{Weave}(0, \varepsilon, \widehat{\tau}_{\text{2zarith}}, \widehat{\mathsf{p}}_0, \mathcal{T}_{-1})$ as depicted in Fig. 4.6a:

$$\begin{aligned}\mathcal{T}_0 = &\left(\mathsf{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 0 & \to & (\varnothing \text{ as } \mathrm{I}_{128}) \parallel (\varnothing) \\ 1 & \to & (\{0\}) \parallel (\{0\}) \parallel (\{0\} \text{ as } \mathrm{I}_{63}) \end{array} \right\}\right) \\ \parallel &\left(\mathsf{switch}(.1.[0:1]) \left\{ \begin{array}{lll} 0 & \to & (\varnothing \text{ as } \mathrm{I}_{128}) \parallel (\varnothing) \\ 1 & \to & (\{0\}) \parallel (\{0\}) \parallel (\{0\} \text{ as } \mathrm{I}_{63}) \end{array} \right\}\right)\end{aligned}$$

After weaving the three remaining memory patterns on resultant memory trees, we finally obtain

the following memory tree, which is depicted in Fig. 4.6b:

$$\mathcal{T}_3 = \mathcal{T}_l \parallel \mathcal{T}_r$$

$$\mathcal{T}_l = \mathsf{switch}(.0.[0:1]) \left\{ \begin{array}{lll} 1 & \to & (\{0,2\}) \parallel (\{0,2\}) \parallel (\{0,2\} \text{ as } I_{63}) \\ 0 & \to & (\{1,3\} \text{ as } I_{128}) \parallel (\{1,3\}) \end{array} \right\}$$

$$\mathcal{T}_r = \mathsf{switch}(.1.[0:1]) \left\{ \begin{array}{lll} 1 & \to & (\{0,3\}) \parallel (\{0,3\}) \parallel (\{0,3\} \text{ as } I_{63}) \\ 0 & \to & (\{1,2\} \text{ as } I_{128}) \parallel (\{1,2\}) \end{array} \right\}$$

At this point, we have integrated all information from memory type and patterns into the memory tree. As seen in Example 4.3, $\mathcal{T}_3$ already has the desired semantics, that is, its evaluation outputs the identifiers of patterns which match the input memory value. △

4.3.3 Decision tree finalization

At this stage, we have woven all patterns into the memory tree. Our memory tree is thus "complete", in that it contains all information from both memory type and patterns. During weaving, the shape of the tree must not be changed: indeed, it must remain synchronized with the memory type for memory patterns to be woven in the right places. Now that weaving is done, however, we can reshape the tree in arbitrary ways as long as semantics are preserved. The goal of this phase is to simplify the memory tree to prepare it for *sequential* code generation. At the end of these simplification passes, we get a *decision tree* which corresponds to "switch nest"-style executable code. As a first step, we "trim" the tree by removing its remaining typing information. We then sequentialize it and keep a single output identifier for each leaf. At any point after trimming, we can apply various classic optimizations, some of which are sketched in Section 4.3.4.

Trimming Since we have explored all patterns, the remaining buds will never be expanded, and can thus be turned into normal leaves. This is done by the following operation:

$$\textsc{Trim}(\mathcal{T}) \triangleq \mathcal{T}\,[(J \text{ as } \widehat{\tau}) \mapsto (J)]$$

From now on, we assume that no buds remain in the tree.

Sequentialization The next, and most important step, is to remove 'Par' nodes to fit a sequential execution model. $\textsc{Seq}(\mathcal{T})$, defined in Fig. 4.10 is the *sequentialized* version of $\mathcal{T}$, i.e., a semantically equivalent tree that does not contain any parallel nodes. Its definition is based on the following description. When we encounter a parallel node, we first *pick* a branch i (based on heuristics, as described in the next section). We then *graft* the remaining branches onto each leaf of $\mathcal{T}_i$. $\textsc{Graft}\left(\mathcal{T}_{\text{parent}}, \mathcal{T}_{\text{child}}\right)$, defined in Fig. 4.10, places $\mathcal{T}_{\text{child}}$ at the leaves of $\mathcal{T}_{\text{parent}}$ and specializes the child tree's leaves by intersecting them with the initial parent leaf. This might result in empty leaves (indicating unreachable code), which can be removed later. Finally, we sequentialize the resulting tree. Note that we sequentialize the remaining trees *after* grafting. Sequentializing remaining branches before grafting would produce a faster compilation algorithm, but give less freedom for heuristics to pick an appropriate branch at each grafting point.

Finalization The very last step to obtain an actual decision tree is to remove every output identifier but the smallest one from each leaf. This reflects the "first pattern wins" semantics of pattern matching. This is done the following operation:

$$\textsc{Finalize}(\mathcal{T}) \triangleq \mathcal{T}\,[(\{j_1, \ldots, j_n\}) \mapsto (\{j_1\})]$$

Note that we never encounter empty leaves at this stage, since the typing judgment for high-level expressions ensures pattern exhaustivity.

$$
\begin{array}{l}
\textsc{Seq}\{ \\
\quad (\mathrm{J}) \longrightarrow (\mathrm{J}) \\
\quad \mathsf{switch}(\widehat{\pi})\{c_0 \to \mathcal{T}_0, \ldots, c_{n-1} \to \mathcal{T}_{n-1}, \_ \to \mathcal{T}'\} \longrightarrow \mathsf{switch}(\widehat{\pi})\{c_0 \to \textsc{Seq}(\mathcal{T}_0), \ldots, c_{n-1} \to \textsc{Seq}(\mathcal{T}_{n-1}), \_ \to \textsc{Seq}(\mathcal{T}')\} \\
\quad \mathcal{T}_0 \parallel \ldots \parallel \mathcal{T}_{n-1} \longrightarrow \textsc{Seq}\left(\textsc{Graft}\left(\mathcal{T}_i, \left\|_{\substack{0 \leqslant i' < n \\ i' \neq i}} \mathcal{T}_{i'}\right.\right)\right) \text{ where } i = \mathbf{pick}(\mathcal{T}_0, \ldots, \mathcal{T}_{n-1}) \\
\}
\end{array}
$$

$$
\textsc{Graft}\left(\mathcal{T}_{\text{parent}}, \mathcal{T}_{\text{child}}\right) \triangleq \mathcal{T}_{\text{parent}}\left[(\mathrm{J}) \mapsto \mathcal{T}_{\text{child}}\left[(\mathrm{J}') \mapsto (\mathrm{J} \cap \mathrm{J}')\right]\right]
$$

Figure 4.10: Sequentialization of memory trees.

Example 4.6 (Running example – Trim and Seq)**.** Figure 4.7a is immediately obtained from Fig. 4.6b by trimming type information from buds. After this step, some redundancy remains, thus we apply an easy simplification to obtain Fig. 4.7b. We are now ready to remove parallel nodes. Let us pick the left branch $\mathcal{T}_l$ as the new parent tree and graft the right branch $\mathcal{T}_r$ with $\textsc{Graft}(\mathcal{T}_l, \mathcal{T}_r)$. All leaves of $\mathcal{T}_l$ are replaced with a copy of $\mathcal{T}_r$ in which leaves' sets are intersected. For instance, the right child $\{1,3\}$ of $\mathcal{T}_l$ is replaced with a $\mathsf{switch}(.1.[0:1])$ whose leaves are $\{1,3\} \cap \{0,2,3\} = \{3\}$ and $\{1,3\} \cap \{1,2\} = \{1\}$, hence Fig. 4.7c. Notice how the semantics of each tree mentioned in this example are unchanged from $\mathcal{T}_3$ obtained after weaving (Example 4.5). △

4.3.4 Optimizations

At this stage, we are able to compile any given pattern matching and memory layout to an equivalent decision tree. However, this decision tree is possibly inefficient, redundant, or both, as we show next in Example 4.7. In this section, we describe how to make the output of our compilation procedure as compact and efficient as possible, by 1) selecting the "best" possible trees during the sequentialization pass, and 2) applying post-processing optimizations to remove dead code after the fact.

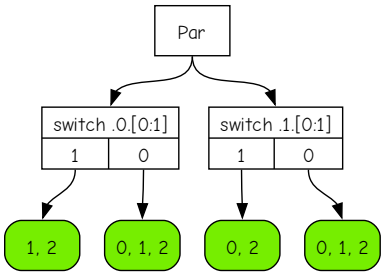
4.3.4.1 Sequentialization heuristics

The first opportunity for optimization in our compilation process arises during sequentialization. In Seq, we non-deterministically pick one of a "Par" node's branches to become the parent of all its siblings. As the following example shows, this choice can dramatically impact the size and efficiency of the resulting decision tree.

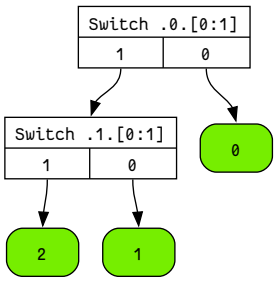
Example 4.7 (Good, bad and ugly decision trees)**.** Consider the following pattern matching in Fig. 4.11 on pairs of Zarith integers, inspired by the list merge example from (Maranget 2008).

```
fn has_bigint(p : ZarithPair) -> bool {
  match p {
    (Large(_), _) => 0,
    (_, Large(_)) => 1,
    _ => 2
  }
}
```

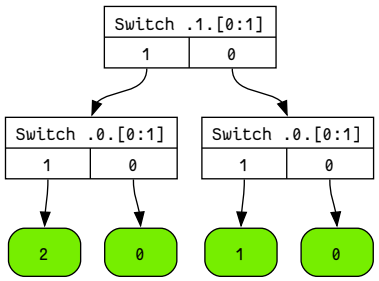
(a) Source code



(b) Memory tree right before sequentialization



(c) Sequentialisation by switching on the field 0 first



(d) Sequentialisation by switching on the field 1 first

Figure 4.11: A simple pattern matching, with different sequentialization heuristics.

After scaffolding the memory type, weaving each memory pattern and trimming, we obtain the memory tree shown in Fig. 4.11b. It contains a single 'Par' Node, and there are two ways to sequentialize it depending on whether we pick the switch node on the first or second field to go first. After sequentialization, finalization and a minor constant folding pass (described later in this section), they yield the two decision trees shown in Figs. 4.11c and 4.11d. These two decision trees are semantically equivalent, but do not correspond to the exact same executable code. Indeed, Fig. 4.11c is better than Fig. 4.11d in two aspects:

- its code size is smaller, since it contains one less switch node;
- it features a "shortcut": while every path in the second tree goes through two switches, the first tree reaches a leaf after only one switch when the first Zarith integer is of the form Large(x).

△

While the difference between the two trees in the previous example seems negligible, some larger programs may benefit from judiciously ordering switch nodes in decision trees, as Scott and Ramsey (2000) show. While runtime performance seems relatively unaffected, static metrics such as code size are more significantly impacted for deep enough pattern nesting, and a handful of (sometimes synthetic) benchmarks are tremendously impacted.

Traditional Heuristics adapted for Memory Trees In our approach, the arrangement of switch nodes in the final decision tree is determined by a series of choices (between children of a "Par" node) in the Seq procedure. Such non-deterministic choices between independent subpatterns are common in all pattern matching compilation approaches. As a result, the problem of deciding the order of switches in a decision tree has been extensively studied since the eighties (Cardelli 1984; Baudinet

and MacQueen 1985; Maranget 1992; Scott and Ramsey 2000), yielding a variety of heuristics tailored to traditional pattern matching compilation approaches. Here, we informally describe the heuristics presented in (Scott and Ramsey 2000) and how they can be adapted to our setting. Consider a parallel node $\mathcal{T} = \mathcal{T}_0 \parallel \ldots \parallel \mathcal{T}_{n-1}$ that we want to sequentialize. We assume that each child tree $\mathcal{T}_i$ is a switch node. This is easy to achieve: if there exists i such that $\mathcal{T}_i$ is a leaf (J_i), we process it by intersecting all other children trees' leaves with J_i; if $\mathcal{T}_i$ is itself a parallel node $\mathcal{T}_0' \parallel \ldots \parallel \mathcal{T}_{n-1}'$, we replace it in $\mathcal{T}$ with its own children; we recursively apply the two previous transformations on every "Par" node's children until fixpoint.

Relevance A switch node is said to be *relevant* to a given pattern identifier j if it is useful to determine whether an input value is accepted or rejected by j, that is, if j appears in some of its branches and not in others. For our approach, a subtree $\mathcal{T}_i$ is relevant to j if some of its leaves contain j and others do not. The relevance heuristic prioritizes trees which are relevant to "early" patterns, that is, to small identifiers. It assigns the score $-j_i$ to each $\mathcal{T}_i$ where j_i is the smallest pattern identifier for which $\mathcal{T}_i$ is relevant, or $-j_{max} - 1$ if it is not relevant to any identifier where j_{max} is the largest pattern identifier.

Small defaults Here, the notion of "defaults" of a switch node refers to pattern identifiers for which it is irrelevant, that is, which appear in every leaf (and therefore act as a "default" for values which are not matched by other patterns). This heuristic minimizes the number of such patterns by assigning the score $-N_i$ to each $\mathcal{T}_i$, where N_i is the number of distinct pattern identifiers which appear in every leaf of $\mathcal{T}_i$.

Fewer child rules This heuristic prioritizes trees which lead to the fewest possible pattern identifiers. It assigns the score $-N_i$ to each $\mathcal{T}_i$, where N_i is the number of pattern identifiers that appear in at least one leaf of $\mathcal{T}_i$.

Small/large branching factor The *branching factor* of a switch node refers to its number of branches, including its default branch if it exists. The small (resp. large) branching factor heuristic assigns the score $-N_i$ (resp. $-N_i$) to each $\mathcal{T}_i$, where N_i is the branching factor of its root switch node.

Arity factor The *arity factor* of a tree refers to the number of distinct locations that it explores. We therefore define the arity factor of a leaf to be 0, of a "Par" node to be its number of children, and of a switch node to be the sum of its branches' arity factors. This heuristic minimizes the number of distinct locations to inspect by assigning to each $\mathcal{T}_i$ a score equal to the negation of its arity factor.

Leaf edges The *leaf edge* of a switch node refers to the number of its children that are leaves. This heuristic prioritizes higher leaf edges by assigning to each $\mathcal{T}_i$ a score equal to its leaf edge.

Failure ("artificial rule" in (Scott and Ramsey 2000)) This heuristic prioritizes trees which never lead to a failure to match any pattern, by assigning the score -1 to every $\mathcal{T}_i$ switch node in which at least one branch is an empty leaf (or, equivalently, a "Par" node having at least one empty leaf child), and 0 to other subtrees.

Left-to-right/Right-to-left Unlike previous heuristics, which were based on the general shape of each switch node $\mathcal{T}_i$, these heuristics measure their discriminant paths, which we denote $\widehat{\pi}_i$. While the precise score assigned to each $\mathcal{T}_i$ depends on the exact set of their discriminant paths, both heuristics prioritize shorter paths by assigning a higher score to $\mathcal{T}_i$ than to $\mathcal{T}_{i'}$ if $\widehat{\pi}_i$ is shorter than $\widehat{\pi}_{i'}$. If two of these trees' discriminant paths share a common prefix $\widehat{\pi}$ but differ in their last operation, the left-to-right heuristic prioritizes the leftmost path, while the right-to-left heuristic prioritizes the rightmost path. For instance, when comparing two subtrees $\mathcal{T}_i$ and $\mathcal{T}_{i'}$ whose discriminant paths are $\widehat{\pi}_i = \widehat{\pi}.k$ and $\widehat{\pi}_{i'} = \widehat{\pi}.k'$, if $k < k'$, then the left-to-right (resp. right-to-left) heuristic will assign a higher score to $\widehat{\pi}_i$ (resp. $\widehat{\pi}_{i'}$).

Towards Heuristics based on Layouts Previous works on pattern matching compilation consider high-level pattern matching and assume that a somewhat uniform memory layout is used. In our setting, we must consider the added complexity of potentially intricate memory layouts. Indeed, the choice of memory representation may dramatically impact the runtime cost of individual switches. For

instance, dereferencing a pointer is more costly than arithmetic and bitwise operations on words. Given the choice between a switch node whose discriminant path contains such costly operations and another switch with only arithmetic and bitwise operations, it seems better to pick the latter rather than the former as the first switch in the final decision tree, performance-wise. Indeed, doing so gives us a chance to get a cheaper path for some memory values.

Furthermore, these heuristics were developed in the 80s-90s. Scott and Ramsey (2000) shows that for most programs, they do not significantly improve runtime performance. However, metrics such as data locality and cache performance are now better predictors of overall performance. Heuristics which have access to memory-level information (for instance, memory paths) may be critical to optimize such aspects. Some initial informal experiments with Ribbit suggest that they are indeed useful.

Here, we briefly explore how precise memory layout specifications could be leveraged to offer more information regarding which switch may lead to a better decision tree. In the same vein as the left-to-right and right-to-left heuristics described earlier, we can measure each switch node's discriminant path and prioritize those with "cheaper" memory paths. As a first approximation of a memory path's "cost", we can use its total number of pointer dereferences. This crude metric could be refined by prioritizing switches on recently accessed locations and not counting the cost of already-accessed pointer dereferences. This would require keeping track of accessed memory paths, which is easy to add to our Seq procedure (for instance by propagating a list $\widehat{\Pi}$ across recursive calls).

Note that in traditional pattern compilation approaches, non-deterministic choices between disjoint patterns are made throughout the compilation process, whereas we perform a single sequentialization pass (Seq) after all patterns have already been woven into the memory tree. As a result, metrics such as code size or average execution path length/cost are already available during sequentialization. This makes it possible to directly measure each subtree and choose the "best" one based on these metrics, at the expense of exponential space requirements. Some initial informal experiments with Ribbit suggest that aggressive hash-consing makes comparing trees directly a viable, albeit slow solution with results similar to heuristic-based sequentialization.

4.3.4.2 Elective surgeries

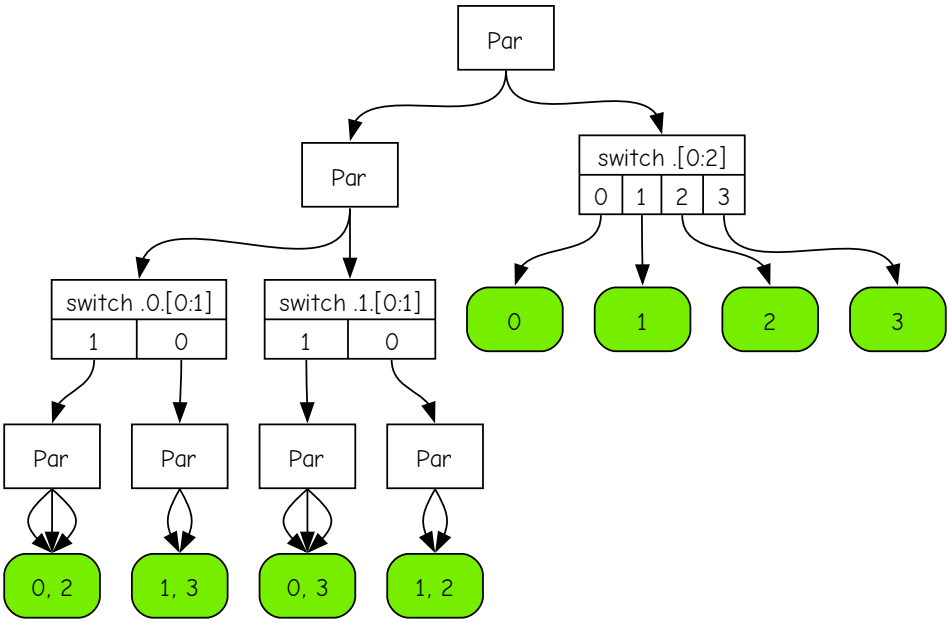
Optimizations on decision trees from literature can be used at any point after trimming. Sequentialization, in particular, might introduce redundant tests or create unreachable branches. Two optimizations are particularly relevant. *Constant folding* propagates information from switches, such as "position $\widehat{\pi}$ contains value c", and uses it to remove redundant switches. *Dead branch elimination* removes branches that lead to empty leaves (i.e., $(\varnothing)$).

Throughout the compilation process, we may also use *sharing* to reduce space requirements, meaning we manipulate Directed Acyclic Graphs rather than trees. In the Ribbit compiler, we achieve this through *hash-consing* (see Chapter 6 for more details).

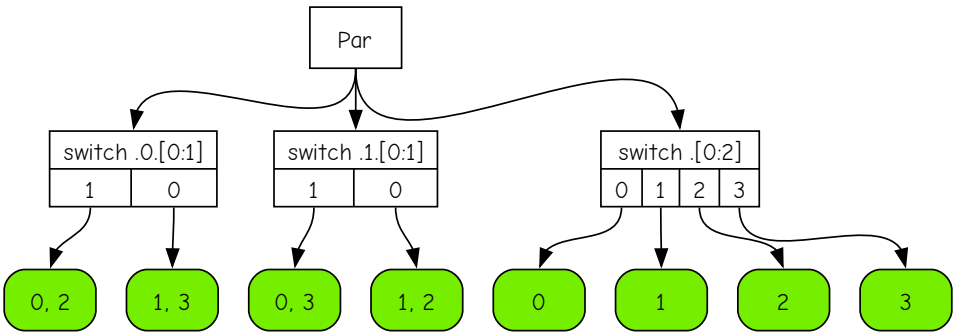
Example 4.8 (Optimizations and sharing for memory trees/DAGs)**.** Consider the following memory layout, which wraps pairs of Zarith integers in a pointer whose two lower bits are reclaimed as a tag, which lets us determine the composition of a pair (e.g., two small integers) without having to dereference it.

$$\&_{64}\left(\widehat{\tau}_{\text{2zarith}}\right) \ltimes [0:2] : \text{split}\,(\varepsilon) \left\{ \begin{array}{lllll} 0 & \text{from} & \langle \text{Small}(\_), \text{Small}(\_) \rangle & \Rightarrow & (0)_2 \\ 1 & \text{from} & \langle \text{Large}(\_), \text{Large}(\_) \rangle & \Rightarrow & (1)_2 \\ 2 & \text{from} & \langle \text{Small}(\_), \text{Large}(\_) \rangle & \Rightarrow & (2)_2 \\ 3 & \text{from} & \langle \text{Large}(\_), \text{Small}(\_) \rangle & \Rightarrow & (3)_2 \end{array} \right\}$$

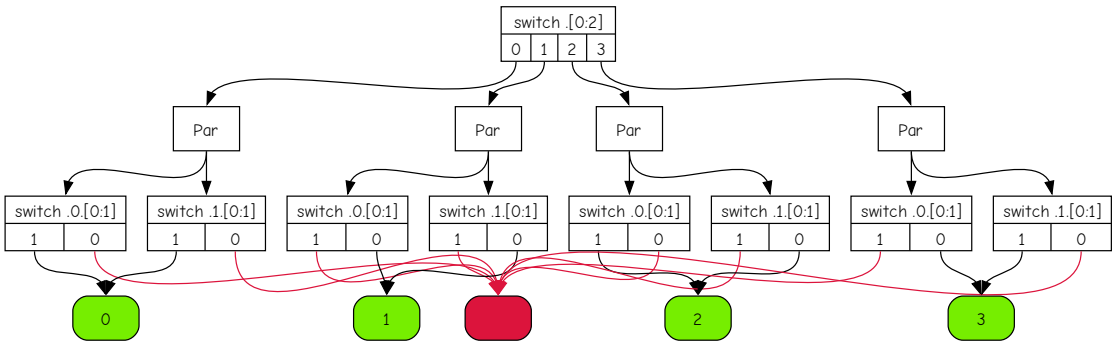
This memory type is redundant by design, leading to empty leaves after sequentialization. After scaffolding it and weaving the memory patterns corresponding to `leq` onto it, we get the following memory tree:



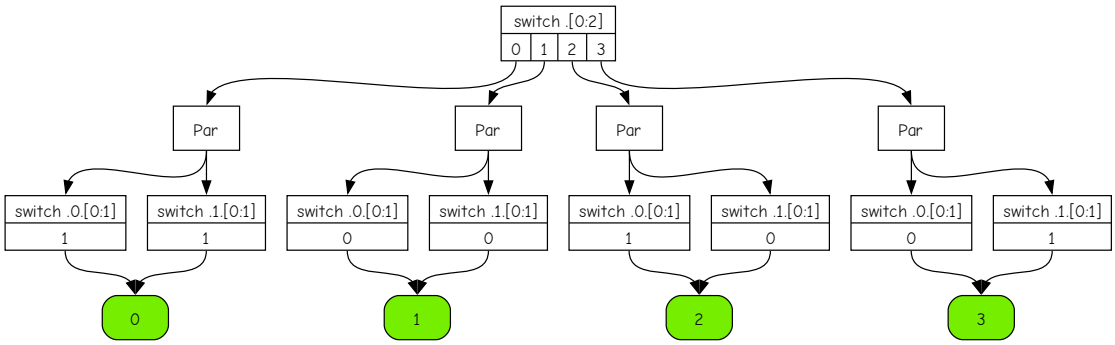
As mentioned in Section 4.3.4.1, before sequentializing this tree, we collapse its nested parallel nodes and remove those whose children are all leaves:



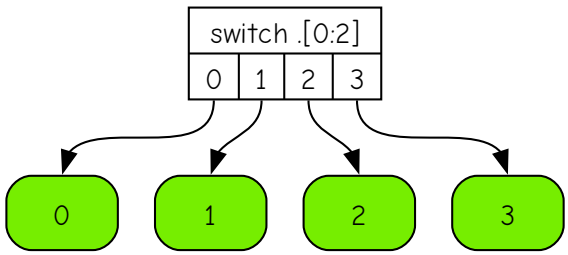
Let us now select the "tag" switch (discriminant path .[0 : 2]) as the root node and graft the rest of the tree to its leaves. We get the following tree, which contains an empty leaf shown in red.



Before attempting to sequentialize the remaining parallel nodes, we can simplify this tree by pruning all "dead paths" (shown in red) which lead to the empty leaf.



The resulting tree consists of a root switch node with four branches in which every path leads to the same leaf. We can therefore simplify it by collapsing each branch into a single leaf, yielding the following decision tree:



△

4.4 Metatheory

We now state and prove the soundness of pattern matching compilation, using the memory-level pattern matching judgment $\blacktriangleright$ as a bridge between high-level pattern matching $\triangleright$ and memory tree evaluation $\blacktriangleright\!\blacktriangleright$. We have already proven in Theorem 3.2 that high-level and memory-level pattern matching (using **pat2mem** with an adequate layout to get memory patterns) are equivalent. The main result of this section is Theorem 4.1, which states that the semantics of the decision tree emitted by our compilation procedure is equivalent to memory-level pattern matching with its input memory patterns.

4.4.1 Tree typing

We restore typing information discarded by our compilation scheme using a separate *tree typing* judgment denoted $\Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau}$ and defined in Fig. 4.12. It ensures that the structure of $\mathcal{T}$ reflects that of the considered memory type $\widehat{\tau}$ when its switch nodes' discriminant paths are prefixed with the current memory path relative to the root memory type $\widehat{\pi}$.

$$\textsc{TreeTTTyVar}\ \frac{(t \mapsto \widehat{\tau}) \in \mathrm{dom}\,(\Delta) \qquad \Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau}}{\Delta, \widehat{\pi} \vdash \mathcal{T} : t}$$

$$\textsc{TreeTLeafWord}\ \Delta, \widehat{\pi} \vdash (J) : \__\ell \qquad \textsc{TreeTLeafConstant}\ \Delta, \widehat{\pi} \vdash (J) : (c)_\ell$$

$$\textsc{TreeTSwitchInt}\ \Delta, \widehat{\pi} \vdash \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{lll} c_0 & \to & (J_0) \\ \dots & \to & \dots \\ c_{n-1} & \to & (J_{n-1}) \\ \_ & \to & (J') \end{array} \right\} : I_\ell$$

$$\textsc{TreeTPointer}\ \frac{\Delta, \widehat{\pi}.* \vdash \mathcal{T} : \widehat{\tau}}{\Delta, \widehat{\pi} \vdash \mathcal{T} : \&_\ell\,(\widehat{\tau})}$$

$$\textsc{TreeTParComposite}\ \frac{\Delta, \widehat{\pi}.\neg r_0 \dots r_{n-1} \vdash \mathcal{T} : \widehat{\tau} \qquad \Delta, \widehat{\pi}.r_i \vdash \mathcal{T}_i : \widehat{\tau}_i}{\Delta, \widehat{\pi} \vdash \mathcal{T} \underset{0 \leqslant i < n}{\|} \mathcal{T}_i : \widehat{\tau} \bowtie_{0 \leqslant i < n} r_i : \widehat{\tau}_i}$$

$$\textsc{TreeTParStruct}\ \frac{\Delta, \widehat{\pi}.i \vdash \mathcal{T}_i : \widehat{\tau}_i}{\Delta, \widehat{\pi} \vdash \mathcal{T}_0 \| \dots \| \mathcal{T}_{n-1} : \{\!\{\widehat{\tau}_0, \dots, \widehat{\tau}_{n-1}\}\!\}}$$

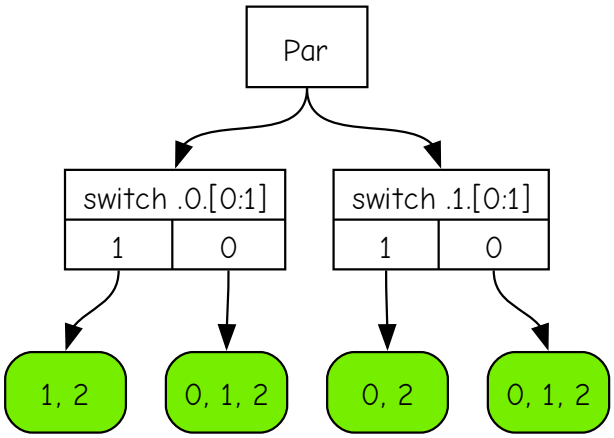
$$\textsc{TreeTBudFragment}\ \Delta, \widehat{\pi} \vdash (J \text{ as } \widehat{\tau}) : (\pi \text{ as } \widehat{\tau})$$

$$\textsc{TreeTGrownFragment}\ \frac{\Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau}}{\Delta, \widehat{\pi} \vdash \mathcal{T} : (\pi \text{ as } \widehat{\tau})}$$

$$\textsc{TreeTSwitchSplit}\ \frac{\Delta, \widehat{\pi} \vdash \mathcal{T}_i : \widehat{\tau}_i}{\Delta, \widehat{\pi} \vdash \underset{0 \leqslant k < N}{\|} \left(\mathsf{switch}(\widehat{\pi}.\widehat{\pi}_k) \left\{ c_{k,i} \to \mathcal{T}_i \;\middle|\; 0 \leqslant i < n \right\} \right) : \mathrm{split}\,(\widehat{\pi}_0, \dots, \widehat{\pi}_{N-1}) \left\{ c_{0,i}, \dots, c_{N-1,i} \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n \right\}}$$

Figure 4.12: Typing judgment for memory trees.

Example 4.9 (Tree typing). Consider the following memory tree from Fig. 4.11b:



We show that it is of type $\widehat{\tau}_{\text{2zarith}}$ with an empty root path:

$$\textsc{TreeTParStruct}\ \dfrac{\textsc{TreeTSwitchSplit}\ \dfrac{\textsc{TreeTLeafConstant}\ \overline{.0 \vdash (\{1,2\}) : (1)_1} \quad \textsc{TreeTLeafConstant}\ \overline{.0 \vdash (\{0,1,2\}) : (0)_1}}{.0 \vdash \begin{array}{l}\text{switch}(.0.[0:1])\{\\ \quad 1 \to (\{1,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} : \widehat{\tau}_{\text{zarith}}}\ \textsc{TreeTGrownFragment}\ \dfrac{}{.0 \vdash \begin{array}{l}\text{switch}(.0.[0:1])\{\\ \quad 1 \to (\{1,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} : (.0\ \text{as}\ \widehat{\tau}_{\text{zarith}})} \qquad \textsc{TreeTSwitchSplit}\ \dfrac{\textsc{TreeTLeafConstant}\ \overline{.0 \vdash (\{0,2\}) : (1)_1} \quad \textsc{TreeTLeafConstant}\ \overline{.0 \vdash (\{0,1,2\}) : (0)_1}}{.1 \vdash \begin{array}{l}\text{switch}(.1.[0:1])\{\\ \quad 1 \to (\{0,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} : \widehat{\tau}_{\text{zarith}}}\ \textsc{TreeTGrownFragment}\ \dfrac{}{.1 \vdash \begin{array}{l}\text{switch}(.1.[0:1])\{\\ \quad 1 \to (\{0,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} : (.1\ \text{as}\ \widehat{\tau}_{\text{zarith}})}}{\varepsilon \vdash \begin{array}{l}\text{switch}(.0.[0:1])\{\\ \quad 1 \to (\{1,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} \parallel \begin{array}{l}\text{switch}(.1.[0:1])\{\\ \quad 1 \to (\{0,2\})\\ \quad 0 \to (\{0,1,2\})\\ \}\end{array} : \widehat{\tau}_{\text{2zarith}}}$$

△

4.4.2 Pattern matching compilation correctness

We now state and prove that each of our compilation steps produces a memory tree (i.e., "progresses") and preserves memory tree typing as well as correct pattern identifiers. As a first result, we show that scaffolding yields a well-typed tree whose evaluation on any well-typed memory value yields the initial identifier set.

Lemma 4.1 (Scaffold correctness)**.** *Let* Δ, ς, J, $\widehat{\pi}$, $\widehat{\tau}$ *and* $\widehat{v}$ *such that*

$$\vDash \Delta \vdash \widehat{\tau} \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau} \qquad \widehat{\mathbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{v})\ \textit{is defined}$$

There exists a memory tree $\mathcal{T}$ *such that*

$$\textsc{Scaffold}_{\Delta}(\mathrm{J}, \widehat{\pi}, \widehat{\tau}) = \mathcal{T} \qquad \Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau} \qquad \varsigma, \widehat{v} \vdash \mathcal{T} \blacktriangleright\!\blacktriangleright \mathrm{J}$$

Proof. Immediate by induction on $\widehat{\tau}$. □

We then prove the correctness of our weaving step. In addition to progress and preservation of tree typing, we show that all pattern identifiers already present in the memory tree are preserved by weaving, and that memory values matched by the woven pattern reach its identifier in tree evaluation. We first extend the typing judgment for memory patterns so as to accept any memory pattern generated by **pat2mem**$(p, \widehat{\tau})$ as a member of the type $\widehat{\tau}$. Indeed, when the high-level pattern p is a wildcard $\_$, **pat2mem** emits a wildcard $\_{|\widehat{\tau}|}$ which is not typed as $\widehat{\tau}$ by the memory typing judgment in its current state. A naive solution would be to add a typing rule accepting all appropriately-sized wildcards (i.e., $\vdash \_{|\widehat{\tau}|} : \widehat{\tau}$ for every $\widehat{\tau}$). However, such a rule would also accept memory patterns which are unsuitable for our purposes – for instance, a memory value of a split type in which all discriminants have been replaced with wildcards, rendering its provenance undistinguishable.

Instead, we follow a finer characterization of memory patterns produced by **pat2mem** for a given $\widehat{\tau}$. It is very similar to the existing memory-level typing judgment, but also allows wildcard memory patterns to occur at toplevel and at fragments, which are exactly the positions where a high-level wildcard pattern may be encountered. We handle the former manually through a separate precondition in Lemma 4.2, and the latter by adding the following rule to the typing judgment for memory patterns:

$$\frac{}{\Delta \vdash \_{|\widehat{\tau}|} : (\pi \text{ as } \widehat{\tau})}\ \textsc{MemTWildcard}$$

Lemma 4.2 (Weave correctness)**.** *Let* $\Delta, \varsigma, J, \widehat{\pi}, \widehat{\tau}, \widehat{v}_\star, \widehat{v}, \widehat{p}, j \notin J$ *and* $\mathcal{T}$ *such that*

$$\vDash \Delta \vdash \widehat{\tau} \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau} \qquad \widehat{\mathbf{focus}_\varsigma}(\widehat{\pi}, \widehat{v}_\star) = \widehat{v} \qquad \Delta \vdash \widehat{p} : \widehat{\tau} \vee \widehat{p} = \_{|\widehat{\tau}|} \qquad \Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau} \qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T} \blacktriangleright\!\blacktriangleright J$$

There exists a memory tree $\mathcal{T}'$ *such that*

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \mathcal{T}' \qquad \Delta, \widehat{\pi} \vdash \mathcal{T}' : \widehat{\tau} \qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright \begin{cases} J \cup \{j\} & \text{if } \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \\ J & \text{otherwise} \end{cases}$$

Proof. If $\widehat{p}$ is a wildcard pattern $\_{|\widehat{\tau}|}$, the WeaveWildcard rule applies:

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \mathcal{T} \left[\begin{array}{lcl} (J) & \mapsto & (J \cup \{j\}) \\ (J \text{ as } \widehat{\tau}') & \mapsto & (J \cup \{j\} \text{ as } \widehat{\tau}') \end{array} \right]$$

Let $\mathcal{T}'$ this woven tree. We always have

$$\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \qquad\qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright J \cup \{j\}$$

Otherwise, we proceed by induction on $\widehat{\tau}$, using tree and memory typing to synchronize $\widehat{p}, \widehat{v}$ and $\mathcal{T}$ with $\widehat{\tau}$. Most cases are immediate; here, we only detail four of them.

Primitive type: $\widehat{\tau} = I_\ell$**.** We have

$$\widehat{p} = (c)_\ell \qquad \widehat{v} = (c')_\ell \qquad \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \iff c = c' \qquad \mathcal{T} = \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{lcl} c_0 & \to & (J_0) \\ \dots & \to & \dots \\ c_{n-1} & \to & (J_{n-1}) \\ \_ & \to & (J') \end{array} \right\}$$

$$J = \begin{cases} J_i & \text{if } \exists i' \in \{0, \dots, n-1\}, c' = c_{i'} \\ J' & \text{otherwise} \end{cases}$$

- If there exists $i \in \{0, \dots, n-1\}$ such that $c = c_i$, the SwitchOldConst and Leaf weaving rules apply:

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{lcl} c_0 & \to & (J_0) \\ \dots & \to & \dots \\ c_i & \to & (J_i \cup \{j\}) \\ \dots & \to & \dots \\ c_{n-1} & \to & (J_{n-1}) \\ \_ & \to & (J') \end{array} \right\}$$

Let $\mathcal{T}'$ this woven tree. If $\widehat{p}$ matches $\widehat{v}$, that is, if $c' = c_i$, we have $J = J_i$ and $\varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright J_i \cup \{j\}$. Otherwise, tree evaluation selects another branch and we have $\varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright J$.

- Otherwise, the SwitchNewConst and Leaf weaving rules apply:

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \mathsf{switch}(\widehat{\pi}) \left\{ \begin{array}{lll} c_0 & \to & (J_0) \\ \dots & \to & \dots \\ c_{n-1} & \to & (J_{n-1}) \\ c & \to & (J' \cup \{j\}) \\ \_ & \to & (J') \end{array} \right\}$$

Let $\mathcal{T}'$ this woven tree. If $\widehat{p}$ matches $\widehat{v}$, that is, if $c' = c$, we have $J = J'$ and $\varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright J' \cup \{j\}$. Otherwise, tree evaluation selects another branch and we have $\varsigma, \widehat{v}_\star \vdash \mathcal{T}' \blacktriangleright\!\blacktriangleright J$.

Fragment type: $\widehat{\tau} = (\pi \textbf{ as } \widehat{\tau}')$. We have $\Delta \vdash \widehat{p} : \widehat{\tau}'$ and $\Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}'$. Suppose that the results holds for $\widehat{\tau}'$. There are two possible cases for $\mathcal{T}$:

TreeTBudFragment: $\mathcal{T}$ is a bud (J as $\widehat{\tau}'$). Since $\widehat{p}$ is not a wildcard memory pattern, we expand the bud during weaving with the BudExpand rule:

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}', \widehat{p}, \mathcal{T}_s)$$

where

$$\mathcal{T}_s = \textsc{Scaffold}_\Delta(J, \widehat{\pi}, \widehat{\tau}')$$

From Lemma 4.1, we have

$$\Delta, \widehat{\pi} \vdash \mathcal{T}_s : \widehat{\tau}' \qquad\qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T}_s \blacktriangleright\!\blacktriangleright J$$

and we conclude using the induction hypothesis.

TreeTGrownFragment: we have $\Delta, \widehat{\pi} \vdash \mathcal{T} : \widehat{\tau}'$ and conclude using the induction hypothesis.

Struct type. Without loss of generality, we only consider two-field structs: $\widehat{\tau} = \{\!\{\widehat{\tau}_0, \widehat{\tau}_1\}\!\}$. We have

$$\widehat{p} = \{\!\{\widehat{p}_0, \widehat{p}_1\}\!\} \qquad \Delta \vdash \widehat{p}_0 : \widehat{\tau}_0 \qquad \Delta \vdash \widehat{p}_1 : \widehat{\tau}_1 \qquad\qquad \widehat{v} = \{\!\{\widehat{v}_0, \widehat{v}_1\}\!\} \qquad \Delta, \varsigma \vdash \widehat{v}_0 : \widehat{\tau}_0 \qquad \Delta, \varsigma \vdash \widehat{v}_1 : \widehat{\tau}_1$$

$$\varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \iff \varsigma \vdash \widehat{p}_0 \blacktriangleright \widehat{v}_0 \wedge \varsigma \vdash \widehat{p}_1 \blacktriangleright \widehat{v}_1 \qquad\qquad \mathcal{T} = \mathcal{T}_0 \parallel \mathcal{T}_1 \qquad \Delta, \widehat{\pi}.0 \vdash \mathcal{T}_0 : \widehat{\tau}_0 \qquad \Delta, \widehat{\pi}.1 \vdash \mathcal{T}_1 : \widehat{\tau}_1$$

$$\varsigma, \widehat{v}_\star \vdash \mathcal{T}_0 \blacktriangleright\!\blacktriangleright J_0 \qquad\qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T}_1 \blacktriangleright\!\blacktriangleright J_1 \qquad\qquad J = J_0 \cap J_1$$

Suppose that the result holds for $\widehat{\tau}_0$ and $\widehat{\tau}_1$. The ParStruct weaving rule applies:

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \textsc{Weave}_\Delta(j, \widehat{\pi}.0, \widehat{\tau}_0, \widehat{p}_0, \mathcal{T}_0) \parallel \textsc{Weave}_\Delta(j, \widehat{\pi}.1, \widehat{\tau}_1, \widehat{p}_1, \mathcal{T}_1)$$

For both fields $k \in \{0, 1\}$, let $\mathcal{T}_k' = \textsc{Weave}_\Delta(j, \widehat{\pi}.k, \widehat{\tau}_k, \widehat{p}_k, \mathcal{T}_k)$. From our induction hypotheses, we have

$$\varsigma, \widehat{v}_\star \vdash \mathcal{T}_k' \blacktriangleright\!\blacktriangleright \begin{cases} J_k \cup \{j\} & \text{if } \varsigma \vdash \widehat{p}_k \blacktriangleright \widehat{v}_k \\ J_k & \text{otherwise} \end{cases}$$

and therefore

$$\varsigma, \widehat{v}_\star \vdash \mathcal{T}_0' \parallel \mathcal{T}_1' \blacktriangleright\!\blacktriangleright \begin{cases} (J_0 \cap J_1) \cup \{j\} & \text{if } \varsigma \vdash \widehat{p} \blacktriangleright \widehat{v} \\ J_0 \cap J_1 & \text{otherwise} \end{cases}$$

The same reasoning applies to composite word types.

Split type. Without loss of generality, we only consider splits with two discriminant positions:

$$\widehat{\tau} = \text{split}\,(\widehat{\pi}_0, \widehat{\pi}_1)\left\{ c_{0,i}, c_{1,i} \text{ from } P_i \Rightarrow \widehat{\tau}_i \;\middle|\; 0 \leqslant i < n \right\}$$

We have $\mathcal{T} = \mathcal{T}_{s0} \parallel \mathcal{T}_{s1}$ where for both $k \in \{0,1\}$,

$$\mathcal{T}_{sk} = \mathsf{switch}(\widehat{\pi}.\widehat{\pi}_k) \left\{ \begin{array}{ll} c_{k,0} & \to \mathcal{T}_0 \\ \dots & \to \dots \\ c_{k,n-1} & \to \mathcal{T}_{n-1} \end{array} \right\}$$

with $\Delta, \widehat{\pi} \vdash \mathcal{T}_i : \widehat{\tau}_i$ for each $i \in \{0, \dots, n-1\}$ (the $\mathcal{T}_i$ are the same in $\mathcal{T}_{s0}$ and $\mathcal{T}_{s1}$). Suppose that the result holds for every $\widehat{\tau}_i$. Since $\widehat{p}$ is of type $\widehat{\tau}$, there exists a unique branch $i \in \{0, \dots, n-1\}$ such that

$$\Delta \vdash \widehat{p} : \widehat{\tau}_i \qquad \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}_0, \widehat{p}) = (c_{0,i})_\ell \qquad \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}_1, \widehat{p}) = (c_{1,i})_\ell$$

(from MemTSplit and VSplit rules). Similarly, since $\widehat{v}$ is of type $\widehat{\tau}$, there exists a unique branch $i' \in \{0, \dots, n-1\}$ such that

$$\Delta, \varsigma \vdash \widehat{v} : \widehat{\tau}_{i'} \qquad \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}_0, \widehat{v}) = (c_{0,i'})_\ell \qquad \widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}_1, \widehat{v}) = (c_{1,i'})_\ell \qquad \varsigma, \widehat{v}_\star \vdash \mathcal{T}_{i'} \blacktriangleright\!\blacktriangleright J$$

The WeaveSplit rule applies: we have

$$\textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}, \widehat{p}, \mathcal{T}) = \mathcal{T}'_{s0} \parallel \mathcal{T}'_{s1}$$

where

$$\mathcal{T}'_{sk} = \mathsf{switch}(\widehat{\pi}.\widehat{\pi}_k) \left\{ \begin{array}{ll} c_{k,0} & \to \mathcal{T}_0 \\ \dots & \to \dots \\ c_{k,i} & \to \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}_i, \widehat{p}, \mathcal{T}_i) \\ \dots & \to \dots \\ c_{k,n-1} & \to \mathcal{T}_{n-1} \end{array} \right\}$$

- If $i = i'$, we have
$$\varsigma, \widehat{v}_\star \vdash \mathcal{T}'_{sk} \blacktriangleright\!\blacktriangleright J'$$
with J' such that
$$\varsigma, \widehat{v}_\star \vdash \textsc{Weave}_\Delta(j, \widehat{\pi}, \widehat{\tau}_i, \widehat{p}, \mathcal{T}_i) \blacktriangleright\!\blacktriangleright J'$$
and we conclude using our induction hypothesis on $\widehat{\tau}_i$.
- Otherwise, $\widehat{p}$ does not match $\widehat{v}$ as they contain distinct constants in at least one discriminant position, and since $\mathcal{T}_{i'}$ is unchanged in the woven tree for all $i' \neq i$, we have
$$\varsigma, \widehat{v}_\star \vdash \mathcal{T}'_{sk} \blacktriangleright\!\blacktriangleright J$$

□

For all subsequent operations, we only need to show that the result of tree evaluation for any well-typed memory value is preserved at each step (typing becomes unnecessary at this stage, and progress is immediate). This is immediate for Trim (the semantics of a bud and of a leaf with the same set of pattern identifiers are exactly the same). We now prove that sequentialization never alters tree semantics:

Lemma 4.3 (Seq correctness)**.** *Let $\mathcal{T}$, ς, $\widehat{v}$ and J such that*

$$\varsigma, \widehat{v} \vdash \mathcal{T} \blacktriangleright\!\blacktriangleright J$$

We have

$$\varsigma, \widehat{v} \vdash \textsc{Seq}(\mathcal{T}) \blacktriangleright\!\blacktriangleright J$$

Proof. We first prove the correctness of Graft, i.e., given a "parent" tree $\mathcal{T}_p$ and a "child" tree $\mathcal{T}_c$ such that

$$\varsigma, \widehat{v} \vdash \mathcal{T}_p \blacktriangleright\!\blacktriangleright J_p \qquad \varsigma, \widehat{v} \vdash \mathcal{T}_c \blacktriangleright\!\blacktriangleright J_c$$

we have

$$\varsigma, \widehat{v} \vdash \textsc{Graft}(\mathcal{T}_p, \mathcal{T}_c) \blacktriangleright\!\blacktriangleright J_p \cap J_c$$

This is immediate by induction on $\mathcal{T}_p$. The result on Seq is then immediate by induction on $\mathcal{T}$. □

We can finally prove our whole compilation approach correct:

Theorem 4.1. *Let* $\Delta, \widehat{\tau}, \varsigma, \widehat{v}, \{\widehat{p}_0, \dots, \widehat{p}_{n-1}\}$ *and* $J \subseteq \{0, \dots, n-1\}$ *such that:*

$$\vDash \Delta \vdash \widehat{\tau} \qquad \Delta \vdash \widehat{p}_j : \widehat{\tau} \qquad \Delta, \varsigma \vdash \widehat{v} : \widehat{\tau} \qquad J = \left\{ j \;\middle|\; 0 \leqslant j < n \wedge \varsigma \vdash \widehat{p}_j \blacktriangleright \widehat{v} \right\}$$

We define the following memory trees according to our compilation algorithm:

$$\mathcal{T}_{-1} = \textsc{Scaffold}_\Delta(\varnothing, \varepsilon, \widehat{\tau}) \qquad \mathcal{T}_j = \textsc{Weave}_\Delta(j, \varepsilon, \widehat{\tau}, \widehat{p}_j, \mathcal{T}_{j-1}) \qquad \mathcal{T}_n = \textsc{Seq}(\textsc{Trim}(\mathcal{T}_{n-1}))$$

We have

$$\varsigma, \widehat{v} \vdash \mathcal{T}_n \blacktriangleright\!\blacktriangleright J$$

Proof. Immediate from Lemmas 4.1 to 4.3. □

4.5 Related work

The problem of compiling pattern matching has been studied since the eighties, starting with Cardelli (1984) and Augustsson (1985). Since then, several approaches have been proposed to compile high-level patterns to efficient decision trees (or other representations such as backtracking automata). However, these existing approaches do not handle custom memory layouts and are geared towards uniform memory representations which closely follow the shape of high-level terms. Such memory layouts are typically found in garbage-collected functional programming languages, which are where pattern matching was originally available (see for instance the OCaml runtime representation, which we present in Section 2.6.1). Adapting existing approaches to our setting is non-trivial: indeed, our memory types (more precisely, the split construct) introduce dependencies between memory locations (for instance, a value behind a pointer may "depend on" this pointer's tag) which do not mesh well with most state-of-the-art compilation approaches.

- Our memory trees are inspired by *AND-OR trees*, which are used to compile pattern matching in Standard ML of New Jersey (MacQueen 2022; Aitken 1992). Similar to memory trees, AND-OR trees consist of leaves, AND nodes representing products (analogous to "Par" nodes) and OR nodes representing a choice between constructors of a sum type (analogous to switch nodes). The procedure for building an AND-OR tree encoding a given match and emitting a decision tree from it is informally described in an obscure research report (Aitken 1992). MacQueen (2022) mentions representing the "concrete shell" of a type, as well as each pattern, as AND-OR trees, then "overlaying" each pattern tree successively onto the type tree, which seems similar to our Scaffold and Weave procedures. Unfortunately, no more details have been published.

- The most seasoned approach to pattern matching compilation (Cardelli 1984; Augustsson 1985; Maranget 2008) uses *pattern matrices* where each row corresponds to a possible pattern and each column to a subterm of these patterns. Therefore, each row can be seen as encoding a conjunction between multiple parts of a given pattern (akin to our "Par" nodes) and each column as encoding a disjunction between different patterns (akin to our switch nodes). The procedure for building a decision tree from a pattern matrix starts with non-deterministically picking a column to inspect. A switch node corresponding to this column's path is then emitted, and its branches are built by recursively compiling the same matrix specialized for each branch. Other representations of high-level patterns for this approach include simple lists of patterns (Baudinet and MacQueen 1985) and sets of subpatterns dubbed *unmatched frontiers* (Scott and Ramsey 2000). This approach is effective in general, but encodes dependencies poorly: adding information such as "column i must be inspected before accessing column column j" to pattern matrices would be rather unwieldy. This is also a problem for pattern matching on GADTs and dependent types. The solution used in the OCaml compiler, for instance, is to force the order of columns to also encode the dependencies, thus preventing optimizations.

  As detailed in Section 4.3.4.1, various heuristics have been developed to pick columns yielding compact and efficient decision trees (Baudinet and MacQueen 1985; Cardelli 1984; Maranget 1992; Aitken 1992; Sestoft 1996). Scott and Ramsey (2000) provides an experimental evaluation of these

heuristics, and shows that their usefulness is rather limited. Maranget (2008) introduces a notion of *necessity* and uses it to define new heuristics. Kosarev, Lozov, and Boulytchev (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.

- Other matrix-based approaches for strict languages produce *backtracking automata*, which are usually less efficient (since a location/subterm may be inspected multiple times) but more compact than decision trees (Fessant and Maranget 2001), or (acyclic) deterministic finite automata (Pettersson 1992). Another approach consists in *partially evaluating* a naive match evaluator for a given list of patterns, then optimizing the resulting program to get a decision tree (Sestoft 1996).
- Pattern matching compilation for lazy languages may use similar techniques (mainly matrix-based) as for strict languages, but is more complex since the semantics chosen for pattern matching is not fixed and affects program termination. Early approaches solve this problem by fixing a left-to-right order of evaluation (Augustsson 1985; Philip Wadler 1987). Laville (1991) defines an alternative semantics based on *partial terms*. This more flexible semantics forms the basis of later approaches (Puel and Suárez 1993; Maranget 1992; Maranget 1994; Sekar, Ramesh, and Ramakrishnan 1995), in which some heuristics are applicable.

4.6 Conclusion

In this chapter, we described how to compile pattern matching, which is a key component of the Ribbit language (and more generally of programs working with ADTs) to efficient decision trees.

However, pattern matching in most languages (and in the Ribbit DSL presented in Chapter 2) is richer than the simplified model handled by our compilation procedure. A crucial missing feature is the ability to use patterns to bind parts of the matched value to variable symbols, in addition to recognizing its shape.

Indeed, this chapter only covered the compilation of a small fragment of the Ribbit language to a simplified target representation (decision trees). In the next chapter, we will provide a detailed compilation approach for the full expression language of the Ribbitulus. To this end, we will define a lower-level and more expressive target representation superseding decision trees, which handles memory allocation, reads and writes on multiple independent memory locations.

The procedures introduced in this chapter are a crucial component of our full compilation approach. Indeed, in addition to pattern matching compilation, our memory tree-based approach is able to emit code which retrieves the provenance of any high-level value solely from its memory representation following a given (possibly heavily mangled) memory layout. As we will see in the next chapter, this is frequently necessary throughout expression compilation: subterm accessors require us to dynamically determine another value's precise high-level provenance.

Chapter 5

Compilation of Valuexpressions

The previous chapter solved the problem of pattern matching compilation with custom memory layouts. We now consider the remaining constructs of the Ribbit language. The most problematic are *pivot expressions*, which combine data constructors and variable accessors with arbitrary memory layouts. Constructors and accessors are simple operations when the underlying memory layout is somewhat similar to high-level values. However, in the context of Ribbit, this is not the case in general: we allow users to specify an arbitrary memory layout for each value. In particular, a pivot expression $(u : \tau \text{ as } \widehat{\tau})$ specifies a memory layout for each variable appearing in u, which may be very different from the existing memory representation of this value. In other words, pivot expressions may introduce implicit data casts between different memory layouts!

In this chapter, we provide a compilation approach for pivot expressions which handles such arbitrary combinations of memory types. We supersede the decision trees defined in the previous chapter with a new target representation in Destination-Passing Style and a finer memory model supporting multiple memory locations as well as explicit allocation, reads and writes. We then combine this new approach with the pattern matching compilation scheme described in the previous chapter into a single compilation procedure which covers the full Ribbit language.

5.1 Motivating Examples

Before formally describing our compilation approach, let us illustrate the problems it solves on several examples. In this chapter, we will use the RISC-V instruction model presented in Section 2.5 and the packed list layout presented in Section 2.4 as running examples to illustrate different parts of our compilation scheme.

Example 5.1 (High-level and memory types for RISC-V instructions.)**.** Recall the `Reg` and `Instr` ADTs and memory layouts presented in Section 2.5, which model a subset of the RISC-V instruction set consisting of four instructions: `Add` (addition of two register operands), `Addi` (addition of a register and an immediate), `Jal` (unconditional jump-and-link) and `Sw` (store a 32-bit word in memory). We model registers as a simple enumeration of the 32 possible registers X_0 to X_{31}, encoded similarly to a C enum on 5 bits:

$$\tau_{\text{reg}} = X_0 \mid \cdots \mid X_{31} \qquad \widehat{\tau}_{\text{reg}} = \text{split}\,(\varepsilon)\left\{ i \text{ from } X_i \Rightarrow (i)_5 \;\middle|\; 0 \leqslant i < 32 \right\}$$

For our particular instruction subset, each of the four possible constructors can be identified using only its opcode stored in the 7 lowest bits of a 32-bit instruction. The Ribbitulus model for these four RISC-V

instructions corresponds to the following high-level and memory types:

$$\begin{aligned}\tau_{\text{riscv}} = \ & \text{Add}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \tau_{\text{reg}} \rangle) \\ & \mid \text{Addi}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \text{I}_{12} \rangle) \\ & \mid \text{Jal}(\langle \tau_{\text{reg}}, \text{I}_{20} \rangle) \\ & \mid \text{Sw}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \text{I}_{12} \rangle)\end{aligned} \qquad \widehat{\tau}_{\text{riscv}} = \text{split}\,(.[0:7]) \left\{ \begin{array}{lll} \texttt{0x33} \text{ from } \text{Add}(\_) & \Rightarrow & \widehat{\tau}_{\text{Add}} \\ \texttt{0x13} \text{ from } \text{Addi}(\_) & \Rightarrow & \widehat{\tau}_{\text{Addi}} \\ \texttt{0x6f} \text{ from } \text{Jal}(\_) & \Rightarrow & \widehat{\tau}_{\text{Jal}} \\ \texttt{0x23} \text{ from } \text{Sw}(\_) & \Rightarrow & \widehat{\tau}_{\text{Sw}} \end{array} \right\}$$

$$\begin{aligned}\widehat{\tau}_{\text{Add}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x33})_7 \\ & \ltimes [7:5] : (.\text{Add.0 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [12:3] : (0)_3 \\ & \ltimes [15:5] : (.\text{Add.1 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [20:5] : (.\text{Add.2 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [25:7] : (0)_7\end{aligned} \qquad \begin{aligned}\widehat{\tau}_{\text{Addi}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x13})_7 \\ & \ltimes [7:5] : (.\text{Addi.0 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [12:3] : (0)_3 \\ & \ltimes [15:5] : (.\text{Addi.1 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [20:12] : (.\text{Addi.2 as } \text{I}_{12})\end{aligned}$$

$$\begin{aligned}\widehat{\tau}_{\text{Jal}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x6f})_7 \\ & \ltimes [7:5] : (.\text{Jal.0 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [12:7] : (.\text{Jal.1.}[11:7] \text{ as } \text{I}_7) \\ & \ltimes [20:1] : (.\text{Jal.1.}[10:1] \text{ as } \text{I}_1) \\ & \ltimes [21:10] : (.\text{Jal.1.}[0:10] \text{ as } \text{I}_{10}) \\ & \ltimes [31:1] : (.\text{Jal.1.}[19:1] \text{ as } \text{I}_1)\end{aligned} \qquad \begin{aligned}\widehat{\tau}_{\text{Sw}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \\ & \ltimes [7:5] : (.\text{Sw.2.}[0:5] \text{ as } \text{I}_5) \\ & \ltimes [12:3] : (2)_3 \\ & \ltimes [15:5] : (.\text{Sw.0 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [20:5] : (.\text{Sw.1 as } \widehat{\tau}_{\text{reg}}) \\ & \ltimes [25:7] : (.\text{Sw.2.}[5:7] \text{ as } \text{I}_7)\end{aligned}$$

We model the RISC-V instruction `sw x1, x2, 42` as the high-level value $v = \text{Sw}(\langle \text{X}_1, \text{X}_2, 42 \rangle)$, and its representation as $\widehat{\tau}_{\text{riscv}}$ is the following memory value:

$$\begin{aligned}\widehat{v} = \ & \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes \boxed{[7:5] : (10)_5} \ltimes [12:3] : (2)_3 \\ & \ltimes [15:5] : (1)_5 \ltimes [20:5] : (2)_5 \ltimes \boxed{[25:7] : (1)_7}\end{aligned}$$

To obtain the previous memory value, we decomposed the 12-bit immediate $42 = \texttt{0b0000 0010 1010}$ into its 5 lowest bits $\boxed{\texttt{0b01010} = 10}$ and 7 highest bits $\boxed{\texttt{0b000 0001} = 1}$. △

One of the problems addressed by our compilation approach is how to compile source value constructors – that is, pivot expressions such as $(v : \tau_{\text{riscv}} \text{ as } \widehat{\tau}_{\text{riscv}})$ – to low-level code which builds a memory value adequately representing the requested value – in this case, $\widehat{v}$. Such value-building code may perform memory allocation, casts from unspecified words to more precise memory structures and initialization of constants at specific positions. For instance, a snippet of low-level code building the memory value $\widehat{v}$ from $\text{Sw}(\langle \text{X}_1, \text{X}_2, 42 \rangle)$ may be:

```
let x = alloc(32); // allocate the necessary space for the memory value
x.[0:5] := 0x23; x.[12:3] := 2; // constant parts of the memory layout
x.[15:5] := 1; x.[20:5] := 2; // register fragments: rs1 = X1, rs2 = X2
x.[25:7] := 1; x.[7:5] := 10; // split immediate 42
```

Figure 5.1: Code building the memory value $\widehat{v}$ representing $\text{Sw}(\langle \text{X}_1, \text{X}_2, 42 \rangle)$ in the root memory location x.

The low-level code shown in Fig. 5.1 is rather straightforward: it is reasonably easy to produce such code through a simple exploration of the desired memory value. Section 5.4.1 will provide a more formal description of such a procedure.

For pivot expressions containing a valuexpression with accessors $x.\pi$ rather than a simple value, the target code also needs to read from the existing memory value x. For instance, consider the pivot $(x.\text{Sw.0} : \tau_{\text{reg}} \text{ as } \widehat{\tau}_{\text{reg}})$, which accesses the first register operand of an Sw instruction stored in x. We only need to extract the 5 bits at offset 15 within x to get the desired memory value: for instance, with the previously built memory value $\widehat{v}$, we have $\overbrace{\textbf{focus}}\,(.[15:5], \widehat{v}) = (1)_5$, which is indeed the $\widehat{\tau}_{\text{reg}}$ representation of the register value X_1. The following low-level code snippet performs this access for any memory value x, storing the result in a newly allocated memory value r:

```
let r = alloc(5); // allocate 5 bits for the extracted register value
r := x.[15:5]; // extract the desired subterm from the value stored in x
```

Figure 5.2: Code extracting the first register operand from the representation of an Sw instruction stored in the memory location x.

Again, this low-level accessor code is immediate from a simple inspection of the memory type $\widehat{\tau}_{\text{riscv}}$. Section 5.4 will provide a formal procedure to compile such simple accessors.

However, other accessors can be trickier to compile. Broadly speaking, the compilation of an arbitrary accessor $(x.\pi : \tau \text{ as } \widehat{\tau})$ may be problematic for two reasons:

- the subterm $.\pi$ may not be stored as a single fragment within the representation of its parent value x (or may be stored as a single fragment using a different memory layout from $\widehat{\tau}$), requiring us to gather multiple pieces before reassembling them into a $\widehat{\tau}$ memory value, as in Example 5.2;
- the actual location of the fragment encoding $.\pi$ within x may depend on the precise provenance of the high-level value represented in x, requiring us to find and inspect split discriminant locations in x to determine its shape and retrieve the representation of $x.\pi$, as in Example 5.3.

Example 5.2 (Scattered immediate in RISC-V: `imm` accessor)**.** Consider the pivot $(x.\text{Sw}.2 : \text{I}_{12} \text{ as } \text{I}_{12})$, which accesses the immediate operand of an Sw instruction *as a standard, consecutive 12-bit primitive*. This piece of data is not immediately accessible in the memory representation of an Sw instruction. Indeed, in $\widehat{\tau}_{\text{Sw}}$, the immediate subterm .Sw.2 is broken down in two pieces stored in separate locations: its 5 lowest bits are stored at memory position $.[7:5]$ and its 7 highest bits at $.[25:7]$. The following low-level code snippet performs this access on a $\widehat{\tau}_{\text{reg}}$ memory value stored in `x`, storing the extracted immediate in a newly allocated location `imm`:

```
let imm = alloc(12); // allocate 12 bits for the rebuilt immediate
imm.[0:5] := x.[7:5]; // 5 lowest bits stored at offset 7 in the instruction
imm.[5:7] := x.[25:7]; // 7 highest bits stored at offset 25 in the instruction
```

Figure 5.3: Code rebuilding the immediate operand from its two pieces extracted from the representation of an Sw instruction stored in the memory location x.

△

So far, value constructors and accessors only required us to allocate memory and initialize it with the contents of various parts of the input (parent) memory value. However, the following example shows that seemingly simple accessors may require more in-depth inspection of the input value and the use of a "switch" construct reminiscent of decision trees.

Example 5.3 (Variable head location in lists)**.** Recall the high-level list type τ_{list} and "packed" memory layout $\widehat{\tau}_{\text{p}}$ (along with the type variable environment Δ_{list}) from Example 3.4:

$$\Delta_{\text{list}} = \{t_{\text{list}} \mapsto \tau_{\text{list}}, t_{\text{p}} \mapsto \widehat{\tau}_{\text{p}}\} \qquad \tau_{\text{list}} = \text{Nil} \mid \text{Cons}(\langle \text{I}_{32}, t_{\text{list}} \rangle)$$

$$\widehat{\tau}_{\text{p}} = \text{split}\,(.[0:2])\,\{$$

$$\begin{array}{lll} 0 \text{ from } \text{Nil} & \Rightarrow & \_{}_{64} \ltimes [0:2] : (0)_2 \\ 1 \text{ from } \text{Cons}(\langle \_, \text{Nil} \rangle) & \Rightarrow & \_{}_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons}.0 \text{ as } \text{I}_{32}) \\ 2 \text{ from } \text{Cons}(\langle \_, \text{Cons}(\_) \rangle) & \Rightarrow & \&_{64}\left(\{\!\{(.\text{Cons}.0 \text{ as } \text{I}_{32}), (.\text{Cons}.1.\text{Cons}.0 \text{ as } \text{I}_{32}), (.\text{Cons}.1.\text{Cons}.1 \text{ as } t_{\text{p}})\}\!\}\right) \ltimes [0:2] : (2)_2 \end{array}$$

$$\}$$

Assume that x is bound to a non-empty list represented as $\widehat{\tau}_{\text{p}}$, and consider the following pivot expression which accesses its first element: $(x.\text{Cons}.0 : \text{I}_{32} \text{ as } \text{I}_{32})$. Depending on the precise provenance of x, its

subterm .Cons.0 may be stored in two different locations: if x represents a list with only one element, then it is a composite 64-bit word whose 32 bits at offset 2 encode the single I_{32} element. However, if x represents a list with more than one element, it is a pointer to a struct whose first field encodes the first element on 32 bits. As a result, in order to extract the I_{32} representation of x.Cons.0, we must first inspect the two lowest bits of x (as indicated by the split discriminant position $.[0:2]$ in its layout $\widehat{\tau}_p$) to identify its provenance, which determines where the subterm .Cons.0 will be stored. The following low-level code snippet performs this access, assuming that the memory location x contains the $\widehat{\tau}_p$ representation of a *non-empty* list:

```
let res = alloc(32); // allocate 32 bits for the extracted element
switch(x.[0:2]) { // identify the provenance of x
  0 -> fail; // Nil: not supposed to happen
  1 -> // Cons(<_, Nil>): extract from composite word
    res := x.[2:32];
  2 -> // Cons(<_, Cons(_)>): extract from pointer and struct
    res := x.*.0;
}
```

Figure 5.4: Code extracting the first element of a non-empty list from its memory representation as $\widehat{\tau}_p$ stored in the memory location x.

△

The four previous examples have illustrated various situations which can arise during compilation of valuexpressions/pivot expressions. As a final motivating example, let us consider a full Ribbitulus expression in which pivots are combined with other syntactical constructs, namely function calls, let-bindings and pattern matching.

Example 5.4 (Full expression with RISC-V values.)**.** Recall the `is_compressible` function from Exhibit 15, which inspects a 32-bit RISC-V instruction to determine whether it can be compressed into a 16-bit instruction according to Waterman et al. (2019). It uses two auxiliary functions `is_nonzero_register` and `is_popular_register`. Both are simple predicates using pattern matching to determine whether a given register is `x0` or not, or whether it is one of the eight "most popular registers" `x8` to `x15`.

Both of these functions return booleans, which we model through the following types:

$$\tau_{\text{bool}} = \text{True} \mid \text{False} \qquad \widehat{\tau}_{\text{bool}} = \text{split}\,(\varepsilon) \left\{ \begin{array}{l} 0 \text{ from False} \Rightarrow (0)_8 \\ 1 \text{ from True} \Rightarrow (1)_8 \end{array} \right\}$$

For simplicity, we assume that the following primitive operations are available: an "AND" operation between booleans denoted $\wedge$, a comparison predicate $x < c$ between a primitive value stored in x and a constant c of the same width, and an equality operator $x = x'$ between two values of the same type.

We model `is_nonzero_register`, `is_popular_register` and `is_compressible` with the function environment Σ and typing environment Γ defined below. Their bodies correspond to the expressions e_c, $e_{0\text{reg}}$ and e_{preg} respectively. Note that or-patterns and variable patterns have been processed into Ribbitulus pattern matching branches by expanding or-patterns into multiple branches and lifting variables

to let-bindings in right-hand-side expressions.

$$\Gamma = \left\{ \begin{array}{lll} \texttt{is\_compressible} & : & (\ \tau_{\text{riscv}} \text{ as } \widehat{\tau}_{\text{riscv}} \ \to\ \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \\ \texttt{is\_nonzero\_register} & : & (\ \tau_{\text{reg}} \text{ as } \widehat{\tau}_{\text{reg}} \ \to\ \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \\ \texttt{is\_popular\_register} & : & (\ \tau_{\text{reg}} \text{ as } \widehat{\tau}_{\text{reg}} \ \to\ \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \end{array} \right\}$$

$$\Sigma = \left\{ \begin{array}{lll} \texttt{is\_compressible} & \mapsto \lambda x.\ & e_c \\ \texttt{is\_nonzero\_register} & \mapsto \lambda x.\ & e_{0\text{reg}} \\ \texttt{is\_popular\_register} & \mapsto \lambda x.\ & e_{\text{preg}} \end{array} \right\} \qquad e_{0\text{reg}} = \text{match}(x) \left\{ \begin{array}{lll} X_0 & \to & (\text{False} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \\ \_ & \to & (\text{True} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \end{array} \right\}$$

$$e_{\text{preg}} = \text{match}(x) \left\{ \begin{array}{lll} X_8 & \to & (\text{True} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \\ \dots & \to & \dots \\ X_{15} & \to & (\text{True} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \\ \_ & \to & (\text{False} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}}) \end{array} \right\}$$

$e_c = \text{match}(x)$ {

$\text{Jal}(\langle X_1, \_ \rangle) \to$ let $n : I_{20}$ as $I_{20} = x.\text{Jal.1}$ in
$(n < 4096 : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}})$

$\text{Add}(\langle \_, \_, \_ \rangle) \to$ let $r_d : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Add.0}$ in
let $r_{s1} : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Add.1}$ in let $b_{s1} : \tau_{\text{bool}}$ as $\widehat{\tau}_{\text{bool}} = \texttt{is\_nonzero\_register}(r_{s1})$ in
let $r_{s2} : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Add.2}$ in let $b_{s2} : \tau_{\text{bool}}$ as $\widehat{\tau}_{\text{bool}} = \texttt{is\_nonzero\_register}(r_{s2})$ in
$(r_d = r_{s1} \wedge b_{s1} \wedge b_{s2} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}})$

$\text{Addi}(\langle \_, \_, \_ \rangle) \to$ let $r_d : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Addi.0}$ in
let $r_s : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Addi.1}$ in let $b : \tau_{\text{bool}}$ as $\widehat{\tau}_{\text{bool}} = \texttt{is\_nonzero\_register}(r_s)$ in
let $n : I_{12}$ as $I_{12} = x.\text{Addi.2}$ in
$(r_d = r_s \wedge b \wedge n < 64 : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}})$

$\text{Sw}(\langle \_, \_, \_ \rangle) \to$ let $r_0 : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Sw.0}$ in let $b_0 : \tau_{\text{bool}}$ as $\widehat{\tau}_{\text{bool}} = \texttt{is\_popular\_register}(r_0)$ in
let $r_1 : \tau_{\text{reg}}$ as $\widehat{\tau}_{\text{reg}} = x.\text{Sw.1}$ in let $b_1 : \tau_{\text{bool}}$ as $\widehat{\tau}_{\text{bool}} = \texttt{is\_popular\_register}(r_1)$ in
let $n : I_{12}$ as $I_{12} = x.\text{Sw.2}$ in
let $n_l : I_5$ as $I_5 = n.[0:5]$ in let $n_h : I_2$ as $I_2 = n.[10:2]$ in
$(b_0 \wedge b_1 \wedge n_l = 0 \wedge n_h = 0 : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}})$

$\_ \to (\text{False} : \tau_{\text{bool}} \text{ as } \widehat{\tau}_{\text{bool}})$

}

The expression e_c, which corresponds to the body of `is_compressible` from Exhibit 15, combines several syntactical constructions. At toplevel, it is a pattern matching expression, which requires us to emit code that recognizes the shape of the input value – this has been covered in Chapter 4, and we will provide a more formal interface in Section 5.3.

Let us now focus on the right-hand side of its branches, for instance the $\text{Sw}(\langle \_, \_, \_ \rangle)$, which matches our value v from Example 5.1. The right-hand side of this branch is an expression which binds the three operands of an Sw instruction – two registers r_0 and r_1 and a 12-bit immediate n. It then uses these extracted values to compute a boolean value determining whether x is a compressible instruction.

From a high-level perspective, extracting these values is a simple task. Indeed, a simple representation for the input value x would simply encode its 12-bit immediate operand at position .Sw.2 as the primitive type I_{12}. The actual memory layout $\widehat{\tau}_{\text{riscv}}$, however, is not so straightforward: since the immediate operand is stored non-consecutively in two separate pieces, we must reassemble them to bind the expected 12-bit immediate to n. In essence, we need to synthesize code manifesting the *isomorphism* between the non-consecutive representation of n and its desired standard immediate representation. △

Our compilation procedure is able to emit the control flow graph shown in Fig. 5.5, which represents low-level code combining a decision tree for the toplevel match with memory reads, writes and allocations that perform the tasks required for its branches.

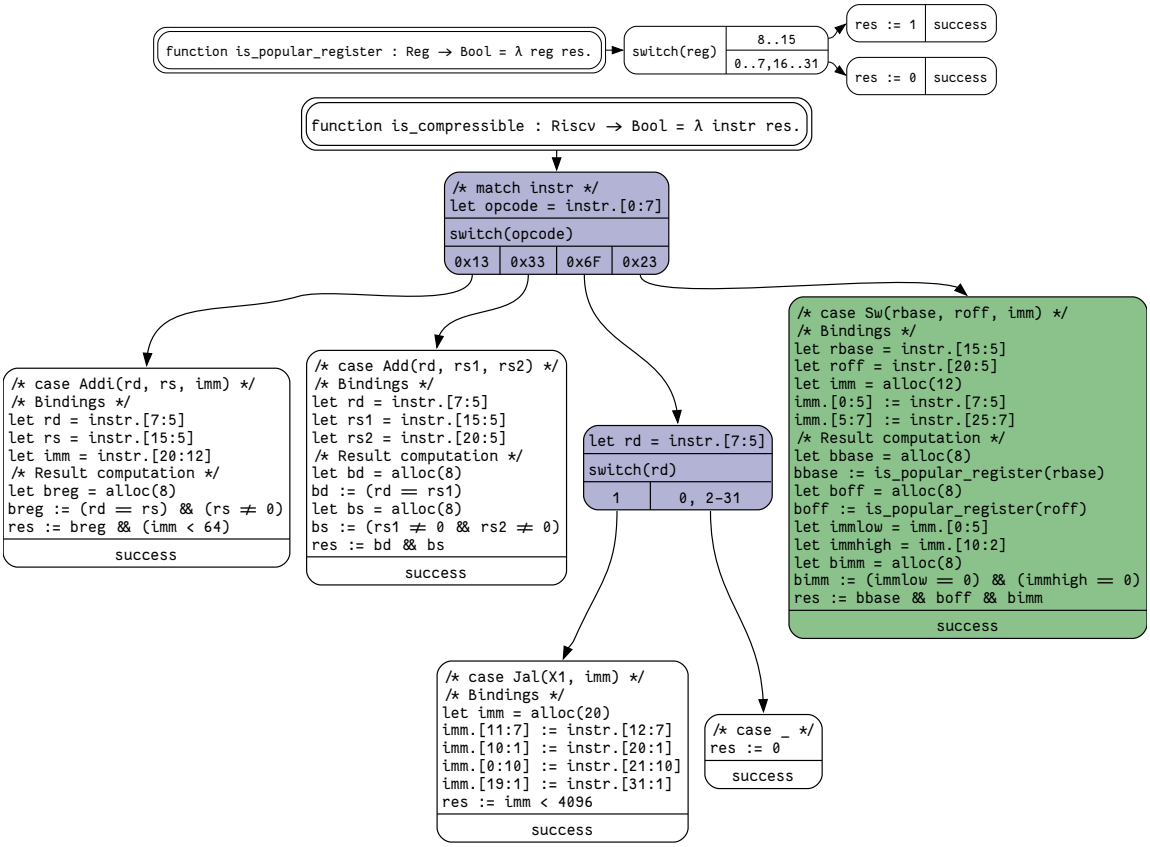


Figure 5.5: Simplified CFGs for **`is_popular_register`** and **`is_compressible`**. From the input `instr`, `is_compressible` identifies the head constructor using the 7 lowest bits, then extracts subterms such as destination and source registers for `Add` or the 12-bit `imm` for `Sw`, and finally computes the boolean result and stores it in `res`.

More precisely, this target program:

- *inspects* the internal representation of an input $\widehat{\tau}_{\text{riscv}}$ value to determine its head constructor (Add, Addi, Jal or Sw), as well as the nested register constructor in Jal;
- *extracts* from this representation all subterms that are bound to variables in the matched pattern. For instance, in the Sw case, the two parts of the 12-bit immediate operand are combined in `imm` in order to reconstruct a value that can be used in a mask;
- *allocates* and *initialises* memory to represent the appropriate values. For instance, the `imm` value just mentioned is first allocated as 12 uninitialized bits, filled, then used to compute the final result.

The rest of this chapter is organized as follows. In Section 5.2, we define a target representation in Destination Passing Style supporting all previously mentioned operations, as well as a detailed execution model. Our compilation procedure consists of several smaller algorithms, each translating a subset of the ribbit language to target expressions. The first of these procedures, Destruct, is defined in Section 5.3 and provides an opaque interface around the pattern matching compilation approach from Chapter 4. In Section 5.4, we introduce necessary tools through simple but incomplete procedures for compiling value constructors and accessors. Section 5.5 details our main compilation algorithms for pivots dubbed Seek and Rebuild. Section 5.6 brings everything together in our toplevel compilation function CompileProg, which handles the whole Ribbitulus in a unified way while also ensuring termination. Finally, in Section 5.7, we state and prove the correctness of our compilation algorithms.

5.2 Target in Destination Passing Style

As we have seen in the previous section, compiled ribbit programs must perform a variety of tasks, including:

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

In this section, we define a target intermediate representation in which each of these tasks is made explicit. We use a Destination-Passing Style (Shaikhha et al. 2017) representation in *A-normal form* (2023). This provides us precise control over memory management and input/output arguments, and could enable further memory improvements, such as using stack allocation when appropriate and applying tail-call modulo cons (Bour, Clément, and Scherer 2021). Another avenue would naturally be to use Continuation-Passing Style (Appel 1992), notably to simplify handling of recursive calls, as we will see in Section 5.6.1. This is in line with numerous compilers for functional languages (Vincent Laviron 2023; Hall, Hammond, et al. 1992) and easily allows moving to SSA representations such as Rust's MIR and LLVM.

5.2.1 Syntax

Our target IR consists of expressions denoted $\tilde{e}$, whose grammar is given in Fig. 5.6. As a convention, target keywords are typeset in a `monospaced font`, and target elements are given a tilde hat (for instance, $\tilde{e}$ is a target expression). Target expressions include calls to functions, which are defined in a global target function environment denoted $\tilde{\Sigma}$ mapping each function symbol $f \in \mathit{FunVars}$ to a lambda-expression with an arbitrary number of arguments $\lambda(x_{\mathrm{in},0}, \ldots, x_{\mathrm{in},n-1}, x_{\mathrm{out}}).\tilde{e}$. Note that sharing is not explicit in the IR, even though we use a control-flow-graph style representation underneath.

$$
\begin{array}{lll}
\tilde{p} ::= \__\ell \underset{0 \leqslant i < n}{\bowtie} [o_i : \ell_i] : \__{\ell_i} \mid \{\!\{ \__{\ell_0}, \ldots, \__{\ell_{n-1}} \}\!\} & & \text{(shallow memory shapes)} \\
\tilde{e} ::= \texttt{success} \mid \texttt{fail} & & \text{(return statements)} \\
\quad \mid \texttt{freeze}(x_{\mathrm{out}\to\mathrm{in}}); \tilde{e} \mid \texttt{let in } x_{\mathrm{in}} = x_{\mathrm{in}}.\widehat{\pi}; \tilde{e} & & \text{(create input locations)} \\
\quad \mid \texttt{let out } x_{\mathrm{out}} = \texttt{alloc}(\ell); \tilde{e} \mid \texttt{let out } x_{\mathrm{out}} = x_{\mathrm{out}}.\widehat{\pi}; \tilde{e} & & \text{(create output locations)} \\
\quad \mid x_{\mathrm{out}} := c; \tilde{e} \mid x_{\mathrm{out}} := \texttt{\&alloc}(\ell); \tilde{e} \mid x_{\mathrm{out}} := x_{\mathrm{in}}; \tilde{e} & & \text{(write to memory)} \\
\quad \mid \texttt{cast } x_{\mathrm{out}} \texttt{ to } \tilde{p}; \tilde{e} \mid \texttt{cast } x_{\mathrm{out}} \texttt{ to } I_\ell; \tilde{e} & & \text{(reinterpret or refine the shape of a memory location)} \\
\quad \mid \texttt{call } f(x_{\mathrm{in},0}, \ldots, x_{\mathrm{in},n-1}, x_{\mathrm{out}}); \tilde{e} & & \text{(function call with } n \text{ input arguments and one destination argument)} \\
\quad \mid \texttt{switch}\,(x_{\mathrm{in}})\,\{c_0 \to \tilde{e}_0, \ldots, c_{n-1} \to \tilde{e}_{n-1}, \_ \to \tilde{e}'\} & & \text{(switch node reading from an input location)} \\
\tilde{\Sigma} : f \mapsto \lambda(x_{\mathrm{in},0} \ldots x_{\mathrm{in},n-1}) x_{\mathrm{out}}.\tilde{e} & & \text{(target function environment)}
\end{array}
$$

Figure 5.6: Target IR grammar.

The essence of destination passing style is that each function takes a *destination* argument which indicates where it should write its result. Similarly, in our IR, *memory locations*, usually denoted x, are identifiers for unaligned pointers. We distinguish between *input* locations ("x_{in}"), which are read-only and correspond to already-built values, and *output* locations ("x_{out}"), which are write-only and correspond to values currently being built. This distinction allows us to formally segregate "analysis" code (i.e., pattern matching) from "building" code (i.e., value constructors). Destination passing style is thus immediately visible in function declarations and calls: the last argument of `call` $f(x_{\mathrm{in},0}, \ldots, x_{\mathrm{in},n-1}, x_{\mathrm{out}})$ is an output memory location x_{out} that should be filled with the result computed by f. For instance, in both CFGs shown in Fig. 5.5, the destination `res` is an output location whose contents at the end of

either function call represent a $\widehat{\tau}_{\text{bool}}$ memory value. Indeed, every `success` statement follows a write instruction of the form res := rhs where rhs designates a boolean value.

Input and output sub-locations are obtained by focusing an existing location with a memory path, for instance with the instruction `let out` $x' = x.\widehat{\pi}$. Additionally, an output location x can be *frozen* to turn it into an input location with `freeze`(x). Constructing values requires memory to write into: either by focusing an existing output location with a memory path (`let out` $x' = x.\widehat{\pi}$), or by claiming a given amount of unused memory (`let out` $x =$ `alloc`(ℓ)). This memory is then filled using `write` instructions, with several kinds of contents: constants denoted c, the contents of an input location, or the address of newly allocated memory of a given size denoted `&alloc`(ℓ).

Newly allocated output locations are filled with an uninitialized word of the form $\__\ell$. To model memory structures such as structs or composite words, they must first be *cast* to an adequate *shallow shape*. Shallow shapes, denoted $\tilde{p}$, are a subset of memory patterns consisting of composite and struct patterns in which every field is a wildcard. Nested memory structures are obtained through several successive casts. Cast instructions are also used to reinterpret integers as primitive values, in situations where they have previously been cast to composite words in order to build their contents as separate bit ranges.

Example 5.5 (Target IR for building an Sw value)**.** The following target code creates a new output location x and fills it with the memory representation of $\text{Sw}(\langle X_1, X_2, 42\rangle)$ as $\widehat{\tau}_{\text{riscv}}$:

$\tilde{e}_{\text{Sw}} =$ `let out` $x =$ `alloc`(32);
`cast` x `to` $\__{32} \ltimes [0:7] : \__7 \ltimes [7:5] : \__5 \ltimes [12:3] : \__3 \ltimes [15:5] : \__5 \ltimes [20:5] : \__5 \ltimes [25:7] : \__7$;
`let out` $x_{\text{opcode}} = x.[0:7]$; $x_{\text{opcode}} :=$ `0x23`; `freeze`(x_{opcode});
`let out` $x_{\text{funct3}} = x.[12:3]$; $x_{\text{funct3}} := 2$; `freeze`(x_{funct3});
`let out` $x_{\text{reg0}} = x.[15:5]$; $x_{\text{reg0}} := 1$; `freeze`(x_{reg0}); (register X_1)
`let out` $x_{\text{reg1}} = x.[20:5]$; $x_{\text{reg1}} := 2$; `freeze`(x_{reg1}); (register X_2)
`let out` $x_{\text{immlow}} = x.[7:5]$; $x_{\text{immlow}} := 10$; `freeze`(x_{immlow}); (5 lowest bits of immediate 42)
`let out` $x_{\text{immhigh}} = x.[25:7]$; $x_{\text{immlow}} := 1$; `freeze`(x_{immhigh}); (7 highest bits of immediate 42)
`freeze`(x);
`success`

This code is driven by the memory layout used for Sw instructions:

$$\widehat{\tau}_{\text{Sw}} = \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [7:5] : (.\text{Sw}.2.[0:5] \text{ as } I_5) \ltimes [12:3] : (2)_3 \\ \ltimes [15:5] : (.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}}) \ltimes [20:5] : (.\text{Sw}.1 \text{ as } \widehat{\tau}_{\text{reg}}) \ltimes [25:7] : (.\text{Sw}.2.[5:7] \text{ as } I_7)$$

We start by allocating 32 bits to store the future $\widehat{\tau}_{\text{Sw}}$ value (since $|\widehat{\tau}_{\text{Sw}}| = 32$) into a new output location x. We then cast these uninitialized 32 bits to a shallow shape which lets us access the location of every constant and every fragment in $\widehat{\tau}_{\text{Sw}}$. We then fill each of these locations with appropriate contents. For each of these components, we start by binding a new output identifier to its location, then write appropriate contents and freeze it after its contents are final. For constants (opcode and funct3 in the RISC-V layout specification), we simply write their values specified in $\widehat{\tau}_{\text{Sw}}$ to their locations. For fragments, which each correspond to an operand of the source Sw instruction, we must first determine the representation of the designated register or immediate. Here, this is easy to achieve: for instance, the representation of the register X_1 as $\widehat{\tau}_{\text{reg}}$ is $(1)_5$ and the representation of the five lowest bits of the immediate 42 as I_5 is 10. △

Example 5.6 (Target IR for rebuilding an Sw immediate)**.** We now define a *function* f_{imm} which, given an input location x_{in} which contains the memory representation as $\widehat{\tau}_{\text{riscv}}$ of an Sw instruction (as obtained, for instance, from the code in Example 5.5), and an output location (a.k.a. *destination*) x_{out} which contains 12 uninitialized bits, fills x_{out} with the representation as I_{12} of the immediate operand (position .Sw.2) of the input value.

Our target function environment will contain the following function binding:

$$f_{\text{imm}} \mapsto \lambda x_{\text{in}} x_{\text{out}}.\tilde{e}_{\text{imm}}$$

with its body $\tilde{e}_{imm}$ defined as follows:

$$\begin{aligned}\tilde{e}_{imm} = \ & \texttt{cast}\ x_{out}\ \texttt{to}\ \_{12} \ltimes [0:5] : \_5 \ltimes [5:7] : \_7; \\ & \texttt{let in}\ x_{inlow} = x_{in}.[7:5];\ \texttt{let out}\ x_{outlow} = x_{out}.[0:5];\ x_{outlow} := x_{inlow};\ \texttt{freeze}(x_{outlow}); \\ & \texttt{let in}\ x_{inhigh} = x_{in}.[25:7];\ \texttt{let out}\ x_{outhigh} = x_{out}.[5:7];\ x_{outhigh} := x_{inhigh};\ \texttt{freeze}(x_{outhigh}); \\ & \texttt{cast}\ x_{out}\ \texttt{to}\ I_{12}; \\ & \texttt{success}\end{aligned}$$

A closed program which builds the representation of $Sw(\langle X_1, X_2, 42\rangle)$ (using the target program $\tilde{e}_{Sw}$ from Example 5.5), then calls f_{imm} to extract its immediate operand would be:

$$\tilde{e}_{Sw};\ \texttt{let out}\ x_{imm} = \texttt{alloc}(12);\ \texttt{call}\ f_{imm}(x, x_{imm});\ \texttt{freeze}(x_{imm});\ \texttt{success}$$

△

Traditional control flow relies on the `switch` construct with a default branch marked by "_", along with `success` and `fail` return statements which do not return any value. The default branch of a `switch` may be omitted, which is equivalent to having a `fail` default branch. Given two target expressions $\tilde{e}$ and $\tilde{e}'$, their sequence $\tilde{e}; \tilde{e}'$ corresponds to $\tilde{e}$ in which every `success` statement has been replaced with $\tilde{e}'$.

Example 5.7 (`is_popular_register` target code)**.** Recall the pattern-matching based function `is_popular_register` from Example 5.4. We translate it to the following target function:

$$\begin{aligned}& f_{reg} \mapsto \lambda x_{in} x_{out}.\texttt{switch}\ (x_{in})\ \{ \\ & \quad 0 \ \to\ x_{out} := 0;\ \texttt{success} \\ & \quad \ldots \ \to\ \ldots \\ & \quad 7 \ \to\ x_{out} := 0;\ \texttt{success} \\ & \quad 8 \ \to\ x_{out} := 1;\ \texttt{success} \\ & \quad \ldots \ \to\ \ldots \\ & \quad 15 \ \to\ x_{out} := 1;\ \texttt{success} \\ & \quad 16 \ \to\ x_{out} := 0;\ \texttt{success} \\ & \quad \ldots \ \to\ \ldots \\ & \quad 31 \ \to\ x_{out} := 0;\ \texttt{success} \\ & \quad \_ \ \to\ \texttt{fail} \\ & \}\end{aligned}$$

As we will see in Section 5.3, this translation from match constructs to target switches is directly based on our pattern matching compilation algorithms presented in Chapter 4. △

5.2.2 Semantics

We now define a small-step evaluation judgment for our target IR. The evaluation judgment, denoted $\tilde{\Sigma} \vdash \tilde{S} \rightsquigarrow \tilde{S}'$ and defined in Figs. 5.9a to 5.9e, uses evaluation states $\tilde{S}$ and a target function environment $\tilde{\Sigma}$ which remains fixed throughout execution/evaluation. It is deterministic (modulo fresh addresses). Elements that appear in gray in an evaluation rule are not necessary for nor affected by the application of this rule.

Evaluation States Target evaluation states, denoted $\tilde{S} = (\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \tilde{e})$, consist of several configuration elements defined in Fig. 5.7, namely a call stack ρ, two environments $\tilde{\sigma}_{in}$ and $\tilde{\sigma}_{out}$ for input and output locations respectively, and a memory store ς. The normal forms are states of the form $(\varnothing, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{success})$; `fail` corresponds to a "stuck" state. We now detail each piece of the evaluation state while looking at the rules that affect them.

$$
\begin{array}{lr}
\tilde{S} ::= (\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \tilde{e}) & \text{(configuration tuple)} \\
\rho ::= \varnothing \mid (\tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \tilde{e}) :: \rho & \text{(call stack)} \\
\tilde{\sigma} : \mathit{Vars} \to \mathit{Addrs} \times \widehat{\mathit{Paths}} & \text{(location binding environment)} \\
\varsigma : \mathit{Addrs} \to \widehat{\mathit{Values}} & \text{(memory store)}
\end{array}
$$

Figure 5.7: Target evaluation states.

Memory Stores The actual memory content is found within the *store* ς which, as in Chapter 3, maps addresses $a \in \mathit{Addrs}$ to concrete memory values $\widehat{v} \in \widehat{\mathit{Values}}$. However, unlike before, we will manipulate memory contents *imperatively*. Indeed, we will allocate and write in memory piece by piece. In particular, a central operation is to insert the value $\widehat{v}$ at position $\widehat{\pi}$ in $\varsigma(a)$. We denote this operation $\varsigma[a.\widehat{\pi} \leftarrow \widehat{v}]$. It relies on an insertion operation denoted $\widehat{\mathbf{insert}}_\varsigma(\varsigma(a), \widehat{\pi}, \widehat{v})$, which writes the new value $\widehat{v}$ at position $\widehat{\pi}$ within the previous value stored at a. The result of insertion is a pair of a new memory store and value $(\varsigma', \widehat{v}')$ such that $\widehat{\mathbf{focus}}_{\varsigma'}(\widehat{\pi}, \widehat{v}') = \widehat{v}$. Its actual definition is rather cumbersome and can be found in Fig. 5.8. It proceeds by induction on memory values in the store, following pointers when necessary. We also define $\widehat{\mathbf{focus}}_\varsigma(\widehat{\pi}, \widehat{v})$, which defines the value at position $\widehat{\pi}$ in $\widehat{v}$ in the store ς.

$$
\frac{(a \mapsto \widehat{v}) \in \varsigma \qquad \widehat{\mathbf{insert}}_\varsigma(\widehat{v}, \widehat{\pi}, \widehat{v}') = (\varsigma', \widehat{v}'')}{\varsigma[a.\widehat{\pi} \leftarrow \widehat{v}'] = \varsigma' \cup \{a \mapsto \widehat{v}''\}}
\qquad\qquad
\frac{\widehat{\mathbf{insert}}_\varsigma(\widehat{v}, \widehat{\pi}, \widehat{v}') = (\varsigma', \widehat{v}'')}{\widehat{\mathbf{focus}}_{\varsigma'}(\widehat{\pi}, \widehat{v}'') = \widehat{v}'}
$$

$$
\begin{array}{l}
\widehat{\mathbf{insert}}_\varsigma(\square, \square, \widehat{v}')\{ \\
\quad \widehat{v} \quad , \ \varepsilon \quad \longrightarrow \quad (\varsigma, \widehat{v}') \\
\quad \widehat{v} \quad , \ .1^{|\widehat{v}|}.\widehat{\pi} \quad \longrightarrow \quad \widehat{\mathbf{insert}}_\varsigma(\widehat{v}, \widehat{\pi}, \widehat{v}') \\
\quad \widehat{v} \underset{0 \leqslant i < n}{\bowtie} [o_i : \ell_i] : \widehat{v}_i \ , \ .(b_{|\widehat{v}|-1} \ldots b_0).\widehat{\pi} \quad \longrightarrow \quad \left(\varsigma', \widehat{v}' \underset{\substack{0 \leqslant i < n \\ i \notin I}}{\bowtie} [o_i : \ell_i] : \widehat{v}_i \right) \\
\qquad\qquad \text{where } I = \left\{ i \;\middle|\; \begin{array}{c} 0 \leqslant i < n \\ \forall j \in \{0, \ldots, \ell_i - 1\}, b_{o_i + j} = 1 \end{array} \right\} \\
\qquad\qquad \text{and } (\varsigma', \widehat{v}') = \widehat{\mathbf{insert}}_\varsigma \left(\widehat{v} \underset{i \in I}{\bowtie} [o_i : \ell_i] : \widehat{v}_i, \widehat{\pi}, \widehat{v}' \right) \\
\quad \&_\ell(a) \quad , \ .*.\widehat{\pi} \quad \longrightarrow \quad (\varsigma' \cup \{a \mapsto \widehat{v}\}, \&_\ell(a)) \text{ where } (\varsigma', \widehat{v}) = \widehat{\mathbf{insert}}_\varsigma(\varsigma(a), \widehat{\pi}, \widehat{v}') \\
\quad \widehat{v} \underset{0 \leqslant i' < n}{\bowtie} r_{i'} : \widehat{v}_{i'} \quad , \ .r_i.\widehat{\pi} \quad \longrightarrow \quad \left(\varsigma', \widehat{v} \ltimes r_i : \widehat{v}'_i \underset{\substack{0 \leqslant i' < n \\ i' \neq i}}{\bowtie} r_{i'} : \widehat{v}_{i'} \right) \text{ where } (\varsigma', \widehat{v}'_i) = \widehat{\mathbf{insert}}_\varsigma(\widehat{v}_i, \widehat{\pi}, \widehat{v}') \\
\quad \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{n-1}\}\!\} \quad , \ .i.\widehat{\pi} \quad \longrightarrow \quad (\varsigma', \{\!\{\widehat{v}_0, \ldots, \widehat{v}_{i-1}, \widehat{v}'_i, \widehat{v}_{i+1}, \ldots, \widehat{v}_{n-1}\}\!\}) \text{ where } (\varsigma', \widehat{v}'_i) = \widehat{\mathbf{insert}}_\varsigma(\widehat{v}_i, \widehat{\pi}, \widehat{v}') \\
\}
\end{array}
$$

Figure 5.8: Write at some position in memory.

Example 5.8. Let us consider $\varsigma = \left\{ a \mapsto \begin{array}{c} \__{32} \ltimes [0:7] : \__7 \ltimes [12:3] : (2)_3 \\ \ltimes [15:5] : (1)_5 \ltimes [20:5] : (2)_5 \\ \ltimes [7:5] : (10)_5 \ltimes [25:7] : (1)_7 \end{array} \right\}$, a store containing a *partially initialized* representation of $\mathrm{Sw}(\langle X_1, X_2, 42 \rangle)$ where the tag, at position $[0:7]$ is missing. We wish to do the insertion $\varsigma[a.[0:7] \leftarrow (\texttt{0x23})_7]$. We have:

$$
\widehat{\mathbf{insert}}_\varnothing(\varsigma(a), .[0:7], (\texttt{0x23})_7) = (\varnothing, \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes \ldots)
$$

hence the following modified store: $\{\mathfrak{a} \mapsto \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes \dots\}$ △

Memory Locations and Bindings Target instructions however never manipulate concrete addresses directly, they are indeed absent from the target language. Instead, each memory address is referred to by its *identifier* x. As mentioned in the previous section, memory locations are either read-only *input locations* x_{in} or write-only *output locations* x_{out}. They are stored in two separate *location environments* $\tilde{\sigma}_{in}$ and $\tilde{\sigma}_{out}$. Both location environments map location identifiers to unaligned pointers of the form $\mathfrak{a}.\widehat{\pi}$, where $\mathfrak{a}$ is an address in the store and $\widehat{\pi}$ is a memory path referring to a precise subterm within the memory value $\varsigma(\mathfrak{a})$.

The creation of new locations is handled by four rules IRENewOutLoc, IRESubOutLoc, IRESubInLoc and IREFreezeLoc, defined in Fig. 5.9a. Newly allocated memory is represented as a fresh address $\mathfrak{a}$ in the store ς mapped to an "uninitialized" word $\__\ell$. Note that the three other rules do not modify the store: they have no effect on concrete memory contents, but instead (dis)allow read/write accesses to precise memory locations. These four rules enforce some degree of separation between input and output locations by maintaining the following invariant:

$$\mathrm{dom}\,(\tilde{\sigma}_{in}) \cap \mathrm{dom}\,(\tilde{\sigma}_{out}) = \varnothing$$

Unfortunately, this is not sufficient for a complete partition of memory between read-only and write-only contents. For instance, if two bindings $x \mapsto \mathfrak{a}.\widehat{\pi}$ and $x' \mapsto \mathfrak{a}.(\widehat{\pi}.\widehat{\pi}')$ are in the output location environment $\tilde{\sigma}_{out}$, after freezing x, we might expect the entire contents of the location $\mathfrak{a}.\widehat{\pi}$ to remain unchanged for the rest of the program's execution. However, this is not guaranteed as x' has not been frozen, which means there is still a way to write to a location within $\mathfrak{a}.\widehat{\pi}$ (namely its subterm at position $\widehat{\pi}'$).

(a) Memory locations

IRENewOutLoc
$$\frac{x \notin \mathrm{dom}\,(\tilde{\sigma}_{in}) \cup \mathrm{dom}\,(\tilde{\sigma}_{out}) \qquad \mathfrak{a} \notin \mathrm{dom}\,(\varsigma)}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{let out}\ x = \texttt{alloc}(\ell); \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out} \cup \{x \mapsto \mathfrak{a}.\varepsilon\}, \varsigma \cup \{\mathfrak{a} \mapsto \__\ell\}, \tilde{e}}$$

IRESubOutLoc
$$\frac{(x \mapsto \mathfrak{a}.\widehat{\pi}_0) \in \tilde{\sigma}_{out} \qquad x' \notin \mathrm{dom}\,(\tilde{\sigma}_{in}) \cup \mathrm{dom}\,(\tilde{\sigma}_{out})}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{let out}\ x' = x.\widehat{\pi}; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out} \cup \{x' \mapsto \mathfrak{a}.\widehat{\pi}_0.\widehat{\pi}\}, \varsigma, \tilde{e}}$$

IREFreezeLoc
$$\frac{x \notin \mathrm{dom}\,(\tilde{\sigma}_{out})}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out} \cup \{x \mapsto \mathfrak{a}.\widehat{\pi}\}, \varsigma, \texttt{freeze}(x); \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in} \cup \{x \mapsto \mathfrak{a}.\widehat{\pi}\}, \tilde{\sigma}_{out}, \varsigma, \tilde{e}}$$

IRESubInLoc
$$\frac{(x \mapsto \mathfrak{a}.\widehat{\pi}_0) \in \tilde{\sigma}_{in} \qquad x' \notin \mathrm{dom}\,(\tilde{\sigma}_{in}) \cup \mathrm{dom}\,(\tilde{\sigma}_{out})}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{let in}\ x' = x.\widehat{\pi}; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in} \cup \{x' \mapsto \mathfrak{a}.\widehat{\pi}_0.\widehat{\pi}\}, \tilde{\sigma}_{out}, \varsigma, \tilde{e}}$$

Figure 5.9: Target IR semantics, memory locations.

Effects on Memory Locations: Writes and Casts We can now define the semantics of operations which write to existing memory locations, namely writes and casts, with the rules shown in Figs. 5.9b and 5.9c. To read from an input location x_{in} with $\tilde{\sigma}_{in}(x_{in}) = \mathfrak{a}.\widehat{\pi}$, we first retrieve the memory value bound to $\mathfrak{a}$ in the store ς, then focus on its subterm at position $\widehat{\pi}$ to get its value. To write to an output location x_{out} with $\tilde{\sigma}_{out}(x_{out}) = \mathfrak{a}.\widehat{\pi}$, we rely on the store update operation $\varsigma[\mathfrak{a}.\widehat{\pi} \leftarrow \widehat{v}]$ defined before. We implement write operations (rules IREWriteConstant, IREWriteAlloc, IREWriteRead) according to the previous principles. For instance, IREWriteRead retrieves the locations of both output and input identifiers, reads the contents of the input location, checks that the output location contains an uninitialized word of the same size, and finally update the store so that the output location receives the input location's contents. Casts are similar to writes: to cast a memory value from an uninitialized word to a shallow shape, we simply interpret this shape as a memory value which we write to the adequate location (IRECastShape). The IRECastInt rule reinterprets composite words as primitive (integer) values similarly to -Fission

rules in various parts of the Ribbitulus. More precisely, it only applies to composite words whose bit ranges partition the total word range and each specify constant contents.

(b) Memory writes

IREWriteConstant

$$\frac{(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{out} \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}, \varsigma(a)) = \__{\ell}}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, x := c; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma[a.\widehat{\pi} \leftarrow (c)_{\ell}], \tilde{e}}$$

IREWriteAlloc

$$\frac{(x \mapsto a_0.\widehat{\pi}) \in \tilde{\sigma}_{out} \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}, \varsigma(a_0)) = \__{\ell_0} \qquad a \notin \mathrm{dom}(\varsigma)}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, x := \texttt{\&alloc}(\ell); \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma[a_0.\widehat{\pi} \leftarrow \&_{\ell_0}(a)] \cup \{a \mapsto \__{\ell}\}, \tilde{e}}$$

IREWriteRead

$$\frac{(x_{in} \mapsto a_{in}.\widehat{\pi}_{in}) \in \tilde{\sigma}_{in} \qquad (x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}_{in}, \varsigma(a_{in})) = \widehat{v}_{in} \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}_{out}, \varsigma(a_{out})) = \__{\ell} \qquad |\widehat{v}_{in}| = \ell}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, x_{out} := x_{in}; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma[a_{out}.\widehat{\pi}_{out} \leftarrow \widehat{v}_{in}], \tilde{e}}$$

(c) Memory casts

IRECastShape

$$\frac{(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{out} \qquad (a \mapsto \widehat{v}) \in \varsigma \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}, \widehat{v}) = \__{\ell} \qquad |\tilde{p}| \leqslant \ell}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{cast } x \texttt{ to } \tilde{p}; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma[a.\widehat{\pi} \leftarrow \tilde{p}], \tilde{e}}$$

IRECastInt

$$\frac{\begin{array}{c}(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{out} \qquad (a \mapsto \widehat{v}) \in \varsigma \qquad \widehat{\textbf{focus}}_{\varsigma}(\widehat{\pi}, \widehat{v}) = \__{\ell} \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : (c_i)_{\ell_i} \\ o_0 = 0 \qquad o_{n-1} + \ell_{n-1} = \ell \qquad o_i = o_{i-1} + \ell_{i-1} \qquad c = \sum_{0 \leqslant i < n} (2^{o_i} \times c_i)\end{array}}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{cast } x \texttt{ to } I_{\ell}; \tilde{e} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma[a.\widehat{\pi} \leftarrow (c)_{\ell}], \tilde{e}}$$

Figure 5.9: (cont'd). Target IR semantics, memory store.

Conditionals Conditionals are simply interpreted as C-style switches. The two rules IRESwitchCase and IRESwitchDefault defined in Fig. 5.9d evaluate switches using the same logic as the decision tree semantics defined in Fig. 4.4. To do so, we read the discriminant value as described above.

Functions and Call Stack Finally, the semantics of functions is modeled via a call stack. The *stack* ρ is either empty $\varnothing$ or consists of a triple $(\tilde{\sigma}'_{in}, \tilde{\sigma}'_{out}, \tilde{e}')$ pushed onto another stack ρ', which we denote $(\tilde{\sigma}'_{in}, \tilde{\sigma}'_{out}, \tilde{e}') :: \rho'$. We use it to evaluate each function call in a fresh context, using two rules IREFunCall and IREReturn defined in Fig. 5.9e. In IREFunCall, we push the current location environments and continuation onto the stack, and evaluate the function body in location environments containing only its arguments. The previous context is restored at the end of the function call with IREReturn. Only the memory store ς persists through function calls and returns.

(d) Switch nodes

IRESwitchCase

$$\frac{(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{in} \qquad (a \mapsto \widehat{v}) \in \varsigma \qquad \exists i \in \{0, \ldots, n-1\}, \widehat{\mathbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{v}) = (c_i)_{\ell}}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{switch}\,(x)\,\{c_0 \to \tilde{e}_0, \ldots, c_{n-1} \to \tilde{e}_{n-1}, \_ \to \tilde{e}'\} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \tilde{e}_i}$$

IRESwitchDefault

$$\frac{(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{in} \qquad (a \mapsto \widehat{v}) \in \varsigma \qquad \widehat{\mathbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{v}) = (c)_{\ell} \qquad c \notin \{c_0, \ldots, c_{n-1}\}}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{switch}\,(x)\,\{c_0 \to \tilde{e}_0, \ldots, c_{n-1} \to \tilde{e}_{n-1}, \_ \to \tilde{e}'\} \rightsquigarrow \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \tilde{e}'}$$

(e) Function calls

IREFunCall

$$\frac{\left(f \mapsto \lambda(x'_0 \ldots x'_{n-1})x'_{out}.\tilde{e}'\right) \in \tilde{\Sigma} \qquad (x_i \mapsto a_i.\widehat{\pi}_i) \in \tilde{\sigma}_{in} \qquad (x_{out} \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{out}}{\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{call}\ f(x_0, \ldots, x_{n-1}, x_{out}); \tilde{e} \rightsquigarrow (\tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \tilde{e}) :: \rho, \varsigma, \{x'_i \mapsto a_i.\widehat{\pi}_i \mid 0 \leqslant i < n\}, \{x'_{out} \mapsto a.\widehat{\pi}\}, \tilde{e}'}$$

IREReturn

$$\tilde{\Sigma} \vdash (\tilde{\sigma}'_{in}, \tilde{\sigma}'_{out}, \tilde{e}') :: \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma, \texttt{success} \rightsquigarrow \rho, \tilde{\sigma}'_{in}, \tilde{\sigma}'_{out}, \varsigma, \tilde{e}'$$

Figure 5.9: (cont'd). Target IR semantics, control flow.

Example 5.9 (Execution of $\mathrm{Sw}(\langle X_1, X_2, 42\rangle)$ value constructor code.)**.** Recall the $\tilde{e}_{Sw}$ target expression from Example 5.5:

$\tilde{e}_{Sw} =$ `let out` $x =$ `alloc`(32);
`cast` x `to` $\_32 \ltimes [0:7] : \_7 \ltimes [7:5] : \_5 \ltimes [12:3] : \_3 \ltimes [15:5] : \_5 \ltimes [20:5] : \_5 \ltimes [25:7] : \_7$;
`let out` $x_{opcode} = x.[0:7]$; $x_{opcode} :=$ `0x23`; `freeze`(x_{opcode});
`let out` $x_{funct3} = x.[12:3]$; $x_{funct3} := 2$; `freeze`(x_{funct3});
`let out` $x_{reg0} = x.[15:5]$; $x_{reg0} := 1$; `freeze`(x_{reg0}); (register X_1)
`let out` $x_{reg1} = x.[20:5]$; $x_{reg1} := 2$; `freeze`(x_{reg1}); (register X_2)
`let out` $x_{immlow} = x.[7:5]$; $x_{immlow} := 10$; `freeze`(x_{immlow}); (5 lowest bits of immediate 42)
`let out` $x_{immhigh} = x.[25:7]$; $x_{immlow} := 1$; `freeze`$(x_{immhigh})$; (7 highest bits of immediate 42)
`freeze`(x);
`success`

We execute it in empty initial environments (we do not show the stack ρ as it remains empty throughout execution):

$\emptyset, \emptyset, \emptyset,$ `let out` x = `alloc`(32); ...

$\rightsquigarrow$ (IRENewOutLoc)

$\emptyset, \{x \mapsto a.\varepsilon\}, \{a \mapsto \_32\},$ `cast` x `to` $\_32 \ltimes [0:7] : \_7 \ltimes [7:5] : \_5 \ltimes \ldots$; ...

$\rightsquigarrow$ (IRECastShape)

$\emptyset, \{x \mapsto a.\varepsilon\}, \{a \mapsto \_32 \ltimes [0:7] : \_7 \ltimes \ldots\},$ `let out` $x_{opcode} = x.[0:7]$; x_{opcode} := `0x23`; `freeze`(x_{opcode}); ...

$\rightsquigarrow$ (IRESubOutLoc)

$\emptyset, \{x \mapsto a.\varepsilon, x_{opcode} \mapsto a.[0:7]\}, \{a \mapsto \_32 \ltimes [0:7] : \_7 \ltimes \ldots\},$ x_{opcode} := `0x23`; `freeze`(x_{opcode}); ...

$\rightsquigarrow$ (IREWriteConstant)

$\emptyset, \{x \mapsto a.\varepsilon, x_{opcode} \mapsto a.[0:7]\}, \{a \mapsto \_32 \ltimes [0:7] : (\texttt{0x23})_7 \ltimes \ldots\},$ `freeze`(x_{opcode}); ...

$\rightsquigarrow$ (IREFreezeLoc)

$\{x_{opcode} \mapsto a.[0:7]\}, \{x \mapsto a.\varepsilon\}, \{a \mapsto \_32 \ltimes [0:7] : (\texttt{0x23})_7 \ltimes \ldots\}, \ldots$

$\rightsquigarrow^{15}$ (IRESubOutLoc, IREWriteConstant, IREFreezeLoc)^5

$$\left\{\begin{array}{l} x_{opcode} \mapsto a.[0:7] \\ x_{funct3} \mapsto a.[12:3] \\ x_{reg0} \mapsto a.[15:5] \\ x_{reg1} \mapsto a.[20:5] \\ x_{immlow} \mapsto a.[5:7] \\ x_{immhigh} \mapsto a.[25:7] \end{array}\right\}, \{x \mapsto a.\varepsilon\}, \left\{a \mapsto \begin{array}{l} \_32 \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [12:3] : (2)_3 \\ \quad \ltimes [15:5] : (1)_5 \ltimes [20:5] : (2)_5 \\ \quad \ltimes [7:5] : (10)_5 \ltimes [25:7] : (1)_7 \end{array}\right\},$$ `freeze`(x); `success`

$\rightsquigarrow$ (IREFreezeLoc)

$\{x \mapsto a.\varepsilon, \ldots\}, \emptyset, \{\ldots\},$ `success`

Notice that the final result bound to a in the store is the same memory value we obtained in Example 5.5. △

Example 5.10. Recall the following target program from Example 5.6, which extracts the immediate operand from a previously built Sw instruction:

$$\tilde{e}_{Sw};\ \texttt{let out}\ x_{imm} = \texttt{alloc}(12);\ \texttt{call}\ f_{imm}(x, x_{imm});\ \texttt{freeze}(x_{imm});\ \texttt{success}$$

with the target function environment $\tilde{\Sigma} = \{f_{imm} \mapsto \lambda x_{in} x_{out}.\tilde{e}_{imm}\}$, where

$$\begin{array}{l} \tilde{e}_{imm} = \texttt{cast}\ x_{out}\ \texttt{to}\ \_12 \ltimes [0:5] : \_5 \ltimes [5:7] : \_7; \\ \qquad \texttt{let in}\ x_{inlow} = x_{in}.[7:5];\ \texttt{let out}\ x_{outlow} = x_{out}.[0:5];\ x_{outlow} := x_{inlow};\ \texttt{freeze}(x_{outlow}); \\ \qquad \texttt{let in}\ x_{inhigh} = x_{in}.[25:7];\ \texttt{let out}\ x_{outhigh} = x_{out}.[5:7];\ x_{outhigh} := x_{inhigh};\ \texttt{freeze}(x_{outhigh}); \\ \qquad \texttt{cast}\ x_{out}\ \texttt{to}\ I_{12}; \\ \qquad \texttt{success} \end{array}$$

Starting from the final state of $\tilde{e}_{Sw}$ execution (see Example 5.9), the execution sequence of the remaining

program `let out` x_{imm} = `alloc`(12); `call` $f_{imm}(x, x_{imm})$; `freeze`(x_{imm}); `success` is:

$\tilde{\Sigma} \vdash \emptyset, \{x \mapsto a.\varepsilon\}, \emptyset, \{a \mapsto \dots\}$, `let out` x_{imm} = `alloc`(12); ...

$\rightsquigarrow$ (IRENewOutLoc)

$\emptyset, \{x \mapsto a.\varepsilon\}, \{x_{imm} \mapsto a_{imm}.\varepsilon\}, \{a \mapsto \dots, a_{imm} \mapsto \_12\}$, `call` $f_{imm}(x, x_{imm})$; ...

$\rightsquigarrow$ (IREFunCall)

... :: $\emptyset, \{x_{in} \mapsto a.\varepsilon\}, \{x_{out} \mapsto a_{imm}.\varepsilon\}, \{a \mapsto \dots, a_{imm} \mapsto \_12\}, \tilde{e}_{imm}$

$\rightsquigarrow^{9}$ (IRECastShape, {IRESubInLoc, IRESubOutLoc, IREWriteRead, IREFreezeLoc}^{2})

$$\dots :: \emptyset, \{x_{in} \mapsto a.\varepsilon, \dots\}, \{x_{out} \mapsto a_{imm}.\varepsilon\}, \left\{a \mapsto \dots, \begin{array}{r} a_{imm} \mapsto \_12 \ltimes [0:5] : (10)_5 \\ \ltimes [5:7] : (1)_7 \end{array}\right\}, \texttt{cast } x_{out} \texttt{ to } I_{12}; \texttt{success}$$

$\rightsquigarrow$ (IRECastInt)

... :: $\emptyset, \{x_{in} \mapsto a.\varepsilon, \dots\}, \{x_{out} \mapsto a_{imm}.\varepsilon\}, \{a \mapsto \dots, a_{imm} \mapsto (42)_{12}\}$, `success`

$\rightsquigarrow$ (IREReturn)

$\emptyset, \{x \mapsto a.\varepsilon\}, \{x_{imm} \mapsto a_{imm}.\varepsilon\}, \{a \mapsto \dots, a_{imm} \mapsto (42)_{12}\}$, `freeze`($x_{imm}$); `success`

$\rightsquigarrow$ (IREFreeze)

$\emptyset, \{x \mapsto a.\varepsilon, x_{imm} \mapsto a_{imm}.\varepsilon\}, \emptyset, \{a \mapsto \dots, a_{imm} \mapsto (42)_{12}\}$, `success`

△

5.3 Pattern matching compilation interface

Now that our target representation is defined, we can describe our compilation approach for the source Ribbit language. It consists of various procedures which compile a subset of Ribbit expressions $e \in \mathit{Exprs}$ to target programs $\tilde{e}$. Our main procedure Compile, which covers the entire Ribbit expression language by wrapping around previous specialized procedures, will be defined in Section 5.6. As a convention, all such compilation procedures are typeset in SmallCaps and defined as algorithms in pseudo-code. Many of these algorithms use Python-style generators with the "**yield**" keyword and "for-each" style loops.

In Algorithm 1, we define our first compilation procedure Destruct, which handles pattern matching expressions. The actual mechanics of pattern matching compilation have already been detailed in Chapter 4: Destruct merely acts as a wrapper around its components **pat2mem**, Scaffold, Weave, Trim and Seq. It also lowers the decision tree emitted by these procedures to equivalent target code. This lets us connect the output of our previous compilation approach to actual memory locations and compiled expressions corresponding to the right-hand sides of pattern matching branches.

Data: Δ a type variable environment
Data: $X = \{x_0, \ldots, x_{n-1}\}$ a list of n root input locations
Data: $\widehat{T} = \{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}$ their memory types
Data: $\{(\{p_{j,0}, \ldots, p_{j,n-1}\}, \tilde{e}_j) \mid 0 \leqslant j < N\}$ a list of N pattern matching branches
Result: Target code determining the first branch j such that every $p_{j,i}$ matches the contents of x_i, then branching to $\tilde{e}_j$

function $\textsc{Destruct}_\Delta(X, \widehat{T}, \{(P_j, \tilde{e}_j) \mid 0 \leqslant j < N\})$:
  // Treat the set of distinct inputs as a single tuple represented as a struct
  $\widehat{\tau} \leftarrow \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$
  $\mathcal{T} := \textsc{Scaffold}_\Delta(\varnothing, \varepsilon, \widehat{\tau})$
  // Assign a fresh output identifier to each memory pattern
  $k := 0$
  $B := \varnothing$
  for $j \in \{0, \ldots, N-1\}$ **do**
    $p_j \leftarrow \langle p_{j,0}, \ldots, p_{j,n-1} \rangle$
    for $(p', \widehat{p}) \in$ **pat2mem**$_\Delta(p_j, \widehat{\tau})$ **do**
      $\mathcal{T} := \textsc{Weave}_\Delta(k, \varepsilon, \widehat{p}, \mathcal{T})$
      $B := B \cup \{(k, \tilde{e}_j)\}$
      $k := k + 1$
  $\mathcal{T} := \textsc{Seq}(\textsc{Trim}(\mathcal{T}))$
  // Lower the decision tree to a target expression
  return $\textsc{TreeToTarget}(X, B, \mathcal{T})$

Algorithm 1: Wrapper around Chapter 4: Destruct.

As we will see in the next sections, we will sometimes need to determine which patterns match *multiple* input values. To this end, Destruct takes as input a list of n memory locations $x_0, \ldots, x_{n-1}$ to inspect, along with their contents' respective memory types $\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}$. Similarly, the N pattern matching branches consist of a list of n high-level patterns $p_{j,0}, \ldots, p_{j,n-1}$ on the left-hand side, and of a target expression $\tilde{e}_j$ on the right-hand side. The semantics of a given branch $\big((p_{j,0}, \ldots, p_{j,n-1}), \tilde{e}_j\big)$ is "if j is the smallest identifier such that each input x_i is matched by its corresponding pattern $p_{j,i}$, then continue execution with $\tilde{e}_j$". Our previous pattern matching compilation approach was designed to inspect a single input value. However, we can easily "trick" it into compiling matches on multiple values by grouping them into a tuple $\langle x_0, \ldots, x_{n-1} \rangle$ and aggregating their memory layouts in a struct $\widehat{\tau} = \{\!\{\widehat{\tau}_0, \ldots, \widehat{\tau}_{n-1}\}\!\}$.

The first compilation steps closely follow the approach described in Chapter 4. We first Scaffold a memory tree $\mathcal{T}$ from the memory layout $\widehat{\tau}$. We process each pattern matching branch j using **pat2mem** to get memory patterns which capture the representations of values matched by this branch. We then Weave each of these memory patterns into $\mathcal{T}$. Note that we weave each memory pattern with a fresh identifier k (which we map back to the right-hand side expression $\tilde{e}_j$ in B). Indeed, it would be incorrect to use the same identifier j for all memory patterns. For instance, if **pat2mem** yielded two memory patterns $\widehat{p}_0 = \{\!\{(0)_{32}, (0)_{32}\}\!\}$ and $\widehat{p}_1 = \{\!\{(1)_{32}, (1)_{32}\}\!\}$, successively weaving $\widehat{p}_0$ and $\widehat{p}_1$ with the same identifier j would yield a tree which accepts incorrect memory values such as $\{\!\{(0)_{32}, (1)_{32}\}\!\}$. Finally, after weaving all patterns, we use Trim and Seq to obtain a decision tree.

The last compilation step, which translates the final decision tree $\mathcal{T}$ to a target expression, is delegated to the TreeToTarget function defined in Fig. 5.10. Given a list of input locations $X = \{x_0, \ldots, x_{n-1}\}$ and a mapping from pattern identifiers to their compiled right-hand side expressions $B = \{(j, \tilde{e}_j) \mid 0 \leqslant j < N\}$, it processes decision tree nodes as follows:

- Leaves $(\{j, \ldots\})$ indicate that the first pattern which matches the input value is the one associated with the identifier j. At this point, the program should continue with the expression corresponding to the right-hand side of this pattern matching branch: we replace the leaf with $\tilde{e}_j$ (Tree2IRLeaf rule). Conversely, empty leaves $(\varnothing)$ indicate that no pattern matches the input value: in this case, we abort execution with `fail` (Tree2IRFail rule).

- Switch nodes $\mathsf{switch}(\widehat{\pi})$ inspect the memory value located at position $\widehat{\pi}$ within the "root input value". Since we grouped our n actual input locations $x_0, \ldots, x_{n-1}$ into a single struct, we necessarily have $\widehat{\pi} = .i.\widehat{\pi}'$ with $i \in \{0, \ldots, n-1\}$. This discriminant path corresponds to the subterm at position $\widehat{\pi}'$ within the input memory location x_i, which we bind to a fresh location identifier x' with a `let in` $x' = x_i.\widehat{\pi}'$ target instruction. We can then emit a target switch expression which inspects this new location x' and whose branches are obtained by recursively transforming the decision node's branches (Tree2IRSwitchDef rule). If the initial switch had no default branch, we use `fail` as the target switch node's default branch (Tree2IRSwitchNoDef rule).

Tree2IRLeaf
$$\textsc{TreeToTarget}(X, \{(j, \tilde{e}_j) \mid 0 \leqslant j < N\}, (\{j, \ldots\})) = \tilde{e}_j$$

Tree2IRFail
$$\textsc{TreeToTarget}(X, B, (\varnothing)) = \texttt{fail}$$

Tree2IRSwitchNoDef
$$\frac{x' \text{ fresh symbol} \qquad \tilde{e}'_j = \textsc{TreeToTarget}(\{x_0, \ldots, x_{n-1}\}, B, \mathcal{T}_j)}{\textsc{TreeToTarget}\left(\{x_0, \ldots, x_{n-1}\}, B, \mathsf{switch}(.i.\widehat{\pi}) \left\{ \begin{array}{lll} c_0 & \to & \mathcal{T}_0 \\ \ldots & \to & \ldots \\ c_{n-1} & \to & \mathcal{T}_{n-1} \end{array} \right\}\right) = \texttt{let in } x' = x_i.\widehat{\pi};\ \texttt{switch}\,(x') \left\{ \begin{array}{lll} c_0 & \to & \tilde{e}'_0 \\ \ldots & \to & \ldots \\ c_{n-1} & \to & \tilde{e}'_{n-1} \\ \_ & \to & \texttt{fail} \end{array} \right\}}$$

Tree2IRSwitchDef
$$\frac{\begin{array}{c} x' \text{ fresh symbol} \\ \tilde{e}'_j = \textsc{TreeToTarget}(\{x_0, \ldots, x_{n-1}\}, B, \mathcal{T}_j) \qquad \tilde{e}' = \textsc{TreeToTarget}(\{x_0, \ldots, x_{n-1}\}, B, \mathcal{T}') \end{array}}{\textsc{TreeToTarget}\left(\{x_0, \ldots, x_{n-1}\}, B, \mathsf{switch}(.i.\widehat{\pi}) \left\{ \begin{array}{lll} c_0 & \to & \mathcal{T}_0 \\ \ldots & \to & \ldots \\ c_{n-1} & \to & \mathcal{T}_{n-1} \\ \_ & \to & \mathcal{T}' \end{array} \right\}\right) = \texttt{let in } x' = x_i.\widehat{\pi};\ \texttt{switch}\,(x') \left\{ \begin{array}{lll} c_0 & \to & \tilde{e}'_0 \\ \ldots & \to & \ldots \\ c_{n-1} & \to & \tilde{e}'_{n-1} \\ \_ & \to & \tilde{e}' \end{array} \right\}}$$

Figure 5.10: From decision trees to target IR expressions.

Let us illustrate this first compilation procedure on a simple pattern matching example.

Example 5.11 (Compilation of `is_popular_register`)**.** Recall the `is_popular_register` function from Example 5.4, which we modelled as the following source function:

$$f_{reg} \mapsto \lambda x.\mathsf{match}(x) \left\{ \begin{array}{lll} X_8 & \to & (\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \ldots & \to & \ldots \\ X_{15} & \to & (\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \_ & \to & (\text{False} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \end{array} \right\}$$

Let x_{out} an output location which we will use as a destination for its result. We admit that the source expression $(\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool})$ translates to the target expression $x_{out} := 1; \texttt{success}$ and $(\text{False} : \tau_{bool} \text{ as } \widehat{\tau}_{bool})$ to $x_{out} := 0; \texttt{success}$. The body of f_{reg} is a simple pattern matching expression, which we compile with the following Destruct call:

$$\textsc{Destruct}\left(\{x\}, \{\widehat{\tau}_{reg}\}, \left\{(\{X_j\}, \tilde{e}_j) \,\middle|\, 0 \leqslant j < 32\right\}\right)$$

where

$$\tilde{e}_j = \begin{cases} x_{out} := 1; \texttt{success} & \text{if } 8 \leqslant j < 16 \\ x_{out} := 0; \texttt{success} & \text{otherwise} \end{cases}$$

The Scaffold, Weave, Trim and Seq steps yield the following decision tree:

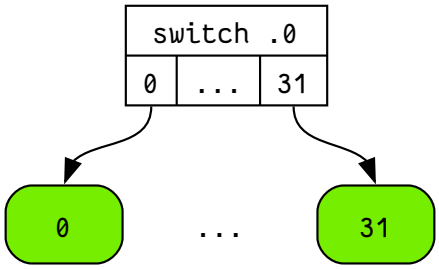


Figure 5.11: Intermediate decision tree in the compilation of `is_popular_register`.

Using TreeToTarget, we replace the switch discriminant .0 with x, each leaf j with the corresponding target expression $\tilde{e}_j$ and get the following target expression:

$$\texttt{switch}\,(x)\left\{\begin{array}{lll} 0 & \to & x_{\text{out}} := 0; \texttt{success} \\ \ldots & \to & \ldots \\ 7 & \to & x_{\text{out}} := 0; \texttt{success} \\ 8 & \to & x_{\text{out}} := 1; \texttt{success} \\ \ldots & \to & \ldots \\ 15 & \to & x_{\text{out}} := 1; \texttt{success} \\ 16 & \to & x_{\text{out}} := 0; \texttt{success} \\ \ldots & \to & \ldots \\ 31 & \to & x_{\text{out}} := 0; \texttt{success} \\ \_ & \to & \texttt{fail} \end{array}\right\}$$

which corresponds to the CFG depicted in Example 5.4. △

5.4 First Naive Approaches for Valuexpressions

We now focus on the compilation of *pivot expressions* of the form $(u : \tau \text{ as } \widehat{\tau})$. Given a destination output location x_{out}, our goal is to emit target code which fills x_{out} with the memory representation of u according to the layout $\widehat{\tau}$. As hinted in Section 5.1, this may raise some unexpected issues: some seemingly simple accessors can require complex manipulation of both input and output memory contents (see for instance Example 5.3).

In this section, we present *incomplete* compilation procedures which handle simple subsets of pivot expressions. These first algorithms provide us with an opportunity to introduce some tooling required for our full compilation approach, presented in the next sections.

5.4.1 Compilation of Value Constructors

As our first foray into the compilation of pivot expressions, we consider their simplest case: $(v : \tau \text{ as } \widehat{\tau})$ where v is a high-level *value* of type τ, without variables. Since values do not contain variable accessors, the memory value $\widehat{v}$ representing v according to $\widehat{\tau}$ is static and largely follows the structure of the memory type $\widehat{\tau}$. Given a destination x_{out}, our goal is to emit a target expression $\tilde{e}$ which allocates, casts and initializes memory so that at the end of its execution, x_{out} contains $\widehat{v}$. This is accomplished with the Construct algorithm defined in Algorithm 2.

Data: Δ a type variable environment
Data: x_{out} an output location
Data: v a high-level value to represent
Data: τ its type
Data: $\widehat{\tau}$ a memory layout agreeing with τ
Result: Target code building the memory representation of v as $\widehat{\tau}$ in x_{out}

1 **function** CONSTRUCT$_\Delta$ $(x_{out}, (v : \tau \text{ as } \widehat{\tau}))$**:**
  // Base case: target value is a constant encoded as a primitive type.
2 **if** $v = c \wedge \widehat{\tau} = I_\ell$ **then return** $x_{out} := c$; *success*
3 **else** // Otherwise, break down the memory type into smaller pieces.
    // Remove splits by selecting the single branch matching v.
4 $\tau' \leftarrow \tau / v$
5 $\{(p, \widehat{\tau}')\} \leftarrow \widehat{\tau} / v$
    // Allocate memory, cast and fill in constant parts as needed for this memory type.
6 $\tilde{e}_{const} \leftarrow$ REFINE $(x_{out}, \_{\mid\widehat{\tau}\mid}, \textbf{shape\_of}(\widehat{\tau}'))$
    // Recursively build memory values representing each fragment (or primitive).
7 frags $\leftarrow$ **shatter**$_\Delta(\widehat{\tau}')$
8 $\tilde{e}_{frags} \leftarrow$ **for** $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in$ *frags* **do**
9 $x_f \leftarrow$ fresh symbol
10 $\tau_f \leftarrow$ **focus** (π_f, τ')
11 $v_f \leftarrow$ **focus** (π_f, v)
12 $\tilde{e}_f \leftarrow$ CONSTRUCT$_\Delta$ $(x_f, (v_f : \tau_f \text{ as } \widehat{\tau}_f))$
13 **yield** *let out* $x_f = x_{out}.\widehat{\pi}_f; \tilde{e}_f;$ *freeze*(x_f)
14 **return** $\tilde{e}_{const}; \tilde{e}_{frags}$

Algorithm 2: CONSTRUCT: a simple compiler for value constructors.

In CONSTRUCT, and in all other algorithms going forward, all equalities between memory types stand for type equivalence up to type variables, for instance we consider the equality "$t = I_\ell$" to be true if $\Delta(t) = I_\ell$. We do not specify *how* to determine whether two types are equal. CONSTRUCT proceeds by inspecting the desired value v and memory type $\widehat{\tau}$ simultaneously to determine which operations are needed to represent v as $\widehat{\tau}$. We assume that at the beginning of execution, x_{out} is an output location whose contents are an adequately sized uninitialized word $\_{\mid\widehat{\tau}\mid}$. If $\widehat{\tau}$ is a primitive type I_ℓ, then from agreement and typing hypotheses, v is necessarily a constant c, and all we have to do is to write it to x_{out} with a single instruction $x_{out} := c$.

Otherwise, we break down the memory type into smaller components. We first process all splits in $\widehat{\tau}$ by specializing it for v (high-level values can indeed be interpreted as patterns). This necessarily yields a single branch $(p, \widehat{\tau}')$, since it is impossible for a value to match more than one branch of the same split.

The next steps follow the resulting split-free memory type $\widehat{\tau}'$ to build the desired memory value. We first consider its "constant" parts which describe concrete memory structures such as words, pointers and structs – these correspond to its shape **shape_of**$_\Delta(\widehat{\tau}')$. In order to mirror the desired memory shape in the contents of x_{out}, we *refine* them using the REFINE function defined in Fig. 5.12 (on line 6).

$\textsc{Refine}_\Delta(x)\{$

$\widehat{p}$	, $\widehat{p}$	$\longrightarrow$	`success`
$\__\ell$	, $(c)_\ell$	$\longrightarrow$	$x :=$ `(c)`$_\ell$; `success`
$\__\ell$	, $\&_\ell(\widehat{p})$	$\longrightarrow$	$x :=$ `&alloc(`$\lvert\widehat{p}\rvert$`)`; $\textsc{Refine}\left(x, \&_\ell\left(\__{\lvert\widehat{p}\rvert}\right), \&_\ell(\widehat{p})\right)$
$\__\ell$	, $\widehat{p} \underset{0\leqslant i<n}{\ltimes} [o_i : \ell_i] : \widehat{p}_i$	$\longrightarrow$	`cast` x `to` $\__\ell \underset{0\leqslant i<n}{\ltimes} [o_i : \ell_i] : \__{\ell_i}$; $\textsc{Refine}\left(x, \__\ell \underset{0\leqslant i<n}{\ltimes} [o_i : \ell_i] : \__{\ell_i}, \widehat{p} \underset{0\leqslant i<n}{\ltimes} [o_i : \ell_i] : \widehat{p}_i\right)$
$\__\ell$	, $\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	$\longrightarrow$	`cast` x `to` $\{\!\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\}\!\}$; $\textsc{Refine}\left(x, \{\!\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\}\!\}, \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}\right)$
$\&_\ell(\widehat{p})$	, $\&_\ell(\widehat{p}')$	$\longrightarrow$	`let out` $x' = x.*$; $\textsc{Refine}(x', \widehat{p}, \widehat{p}')$
$\widehat{p} \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{p}_i$	, $\widehat{p}' \underset{0\leqslant i<n}{\ltimes} r_i : \widehat{p}'_i$	$\longrightarrow$	`let out` $x' = x.\neg r_0 \ldots \neg r_{n-1}$; `let out` $x_0 = x.r_0$; $\ldots$; `let out` $x_{n-1} = x.(n-1)$; $\textsc{Refine}(x', \widehat{p}, \widehat{p}')$; $\textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0)$; $\ldots$; $\textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$
$\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	, $\{\!\{\widehat{p}'_0, \ldots, \widehat{p}'_{n-1}\}\!\}$	$\longrightarrow$	`let out` $x_0 = x.0$; $\ldots$; `let out` $x_{n-1} = x.(n-1)$; $\textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0)$; $\ldots$; $\textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$
$\widehat{p}$	, $\widehat{p}'$	$\longrightarrow$	`fail`

$\}$

Figure 5.12: $\textsc{Refine}(x, \widehat{p}_{\text{old}}, \widehat{p}_{\text{new}})$ – Memory shape refinement instructions.

Refine takes as inputs an output location x, a memory shape $\widehat{p}_{\text{old}}$ which describes the initial contents of x, and a more precise shape $\widehat{p}_{\text{new}}$. It emits memory allocation, cast and write instructions which capture precisely the *difference* between $\widehat{p}_{\text{old}}$ and $\widehat{p}_{\text{new}}$. For instance, to go from an uninitialized word $\__\ell$ to a pointer $\&_\ell(\widehat{p})$, we first allocate memory and store its address in x, then recursively Refine its uninitialized contents to $\widehat{p}$. After executing these instructions, which will be shown in green in the following examples, the contents of x conform to the desired shape $\widehat{p}_{\text{new}}$.

Example 5.12 (Refine the shape of an Sw instruction.)**.** Recall the τ_{riscv} value $v = \text{Sw}(\langle X_1, X_2, 42\rangle)$ from Example 5.1. Our goal is to emit code which builds its memory representation as $\widehat{\tau}_{\text{riscv}}$ in the output location x_{out} with the following Construct call:

$$\textsc{Construct}(x_{\text{out}}, (\text{Sw}(\langle X_1, X_2, 42\rangle) : \tau_{\text{riscv}} \text{ as } \widehat{\tau}_{\text{riscv}}))$$

The specialization of $\widehat{\tau}_{\text{riscv}}$ for v yields the memory type $\widehat{\tau}_{\text{Sw}}$. Assuming that x_{out} initially contains an empty 32-bit word $\__{32}$, the first step is to refine its shape to that of $\widehat{\tau}_{\text{Sw}}$, which is:

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

We are able to do so with a single cast instruction, which is emitted by $\textsc{Refine}(x_{\text{out}}, \__{32}, \textbf{shape\_of}(\widehat{\tau}_{\text{Sw}}))$:

`cast` x_{out} `to` $\__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [7:5] : \__5 \ltimes [12:3] : (2)_3 \ltimes [15:5] : \__5 \ltimes [20:5] : \__5 \ltimes [25:7] : \__7$

△

The next step is to emit target code which fleshes out the skeleton created by Refine with the "variable" parts of $\widehat{\tau}'$. These parts correspond to fragments and primitive types appearing in $\widehat{\tau}'$, which we gather on line 7 as a set of triplets of the form $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f)$ using the **shatter** function (which was defined in Fig. 3.13, Section 3.1). Unlike constant parts whose memory contents are solely determined by the memory type, the contents of these variable parts also depend on the input value v. Indeed, each triplet $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f)$ specifies that the memory value at position $\widehat{\pi}_f$ within the root destination x_{out} should represent the subterm at position π_f in the high-level value v as the memory layout $\widehat{\tau}_f$. Our goal is now to emit code, gathered into $\tilde{e}_{\text{frags}}$, which builds the memory representation of these subterms in their appropriate locations. For each fragment (or primitive) $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f)$, we proceed as follows (lines 8 to 13):

1. Define a new output location x_f which will be the destination of this fragment. It corresponds to the position $\widehat{\pi}_f$ within the current destination x_{out}.
2. Recursively call Construct to emit code $\tilde{e}_f$ which builds the representation of the desired subterm **focus** (π_f, v) as $\widehat{\tau}_f$ in x_f.

3. Freeze x_f since there is no need to modify its contents after this point.

In examples, we will display the target instructions emitted by these steps in a light blue box.

Note the use of Python-style generators with `for` and `yield` keywords on lines 8 and 13. In general, we will reuse this functional iterator-based style throughout this chapter to build lists of target expressions.

Example 5.13 (Build variable parts of an Sw instruction)**.** The target expression shown in Example 5.12 handled all constant parts of $\widehat{\tau}_{\text{Sw}}$. We now resume the execution of this Construct call after line 6. We emit target code which handles all variable parts of $\widehat{\tau}_{\text{Sw}}$ to build the representation of the high-level value $v = \text{Sw}(\langle X_1, X_2, 42\rangle)$. We have

$$\textbf{shatter}(\widehat{\tau}_{\text{Sw}}) = \left\{ \begin{array}{l} (.[7:5] \mapsto .\text{Sw}.2.[0:5] \text{ as } I_5), \\ (.[15:5] \mapsto .\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}}), \\ (.[20:5] \mapsto .\text{Sw}.1 \text{ as } \widehat{\tau}_{\text{reg}}), \\ (.[25:7] \mapsto .\text{Sw}.2.[5:7] \text{ as } I_7) \end{array} \right\}$$

Let us first deal with the first immediate fragment $(.[7:5] \mapsto .\text{Sw}.2.[0:5] \text{ as } I_5)$. The corresponding subterm in our input value v is

$$\textbf{focus}\,(.\text{Sw}.2.[0:5], v) = \textbf{focus}\,(.[0:5], 42) = 10$$

which is of type

$$\textbf{focus}\,\left(.\text{Sw}.2.[0:5], \text{Sw}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}\rangle)\right) = \textbf{focus}\,(.[0:5], I_{12}) = I_5$$

We perform the following recursive Construct call to build its representation as I_5, where $x_{2,05}$ is a fresh location symbol:

$$\textsc{Construct}\,(x_{2,05}, (10 : I_5 \text{ as } I_5))$$

Since 10 is a primitive value which we want to represent as the primitive memory type I_5, this call is a "base case" of Construct and yields the target expression $x_{2,05} := 10; \texttt{success}$. To build this fragment, we therefore emit the following target expression:

$$\texttt{let out } x_{2,05} = x_{\text{out}}.[7:5];\ x_{2,05} := 10;\ \texttt{freeze}(x_{2,05});\ \texttt{success}$$

We now process the first register operand represented by the triplet $(.[15:5] \mapsto .\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})$. The corresponding subterm in our input value v is $\textbf{focus}\,(.\text{Sw}.0, v) = X_1$, which is of type $\textbf{focus}\,\left(.\text{Sw}.0, \text{Sw}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}\rangle)\right) = \tau_{\text{reg}}$. We perform the following recursive Construct call to build its representation as $\widehat{\tau}_{\text{reg}}$, where x_0 is a fresh location symbol:

$$\textsc{Construct}\,\left(x_0, (X_1 : \tau_{\text{reg}} \text{ as } \widehat{\tau}_{\text{reg}})\right)$$

We first specialize $\widehat{\tau}_{\text{reg}}$ for X_1, yielding the memory type $(1)_5$. This constant type does not contain any primitive or fragment, therefore the code emitted by this recursive Construct call is:

$$\textsc{Refine}(x_0, \__5, (1)_5) = x_0 := 1;\ \texttt{success}$$

To build this fragment, we finally emit the following target expression:

$$\texttt{let out } x_0 = x_{\text{out}}.[15:5];\ x_0 := 1\ ;\ \texttt{freeze}(x_0);\texttt{success}$$

The two other fragments are handled similarly. △

The final target expression emitted by $\textsc{Construct}_\Delta(x_{\text{out}}, (v : \tau \text{ as } \widehat{\tau}))$ on line 14 is the sequence of $\tilde{e}_{\text{const}}$, which refines the contents of x_{out} to build the constant parts of $\widehat{\tau}$, and of $\tilde{e}_{\text{frags}}$, which recursively builds the variable parts of $\widehat{\tau}$ in their respective locations.

Example 5.14 (Construct the representation of an Sw value)**.** Using the target expressions shown in Examples 5.12 and 5.13, we finally obtain the following target code:

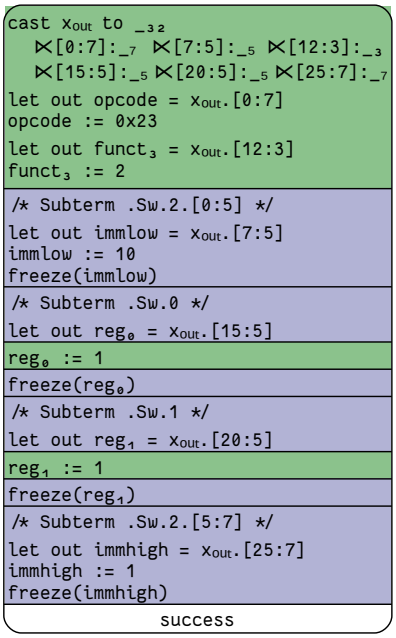


Figure 5.13: Final code to construct the RISC-V $\mathrm{Sw}(\langle X_1, X_2, 42\rangle)$ value according to $\widehat{\tau}_{\mathrm{Sw}}$.

△

5.4.2 Compilation of Accessors: A First Attempt

The Construct algorithm presented in the previous subsection is able to emit target code which builds the memory representation of any value as any suitable memory layout. However, most of the complexity associated with compilation of pivot expressions comes from valuexpressions which combine constant values with variable *accessors* of the form $x.\pi$. As Section 5.1 demonstrates, seemingly simple accessors can require extensive manipulation of both input and output values. Our general compilation approach for arbitrary pivot expressions, which we will present in Section 5.5, is rather complex and relies on several tools to handle different aspects of valuexpressions and memory layouts. We now introduce the basic concepts used to handle accessors in our full approach through a simple, but incomplete procedure.

Consider an accessor $x.\pi$, where x is known to be bound to a value v of type τ whose memory representation $\widehat{v}$ follows the layout $\widehat{\tau}$. A first intuition to compile this accessor could be to look for a position $\widehat{\pi}$ at which $\widehat{v}$ contains the memory representation of the subterm **focus** (π, v). For instance, consider the RISC-V "store word" instruction $\widehat{v} = \mathrm{Sw}(\langle v_{\mathrm{reg0}}, v_{\mathrm{reg1}}, v_{\mathrm{imm}}\rangle)$ represented using the memory type $\widehat{\tau}_{\mathrm{Sw}}$ from Example 5.1. The memory representation of its first register operand v_{reg0} is stored at position $.[15:5]$ within $\widehat{v}$, since this position corresponds to the following fragment in $\widehat{\tau}_{\mathrm{Sw}}$:

$$\widehat{\textbf{focus}}\ (.[15:5], \widehat{\tau}_{\mathrm{Sw}}) = (.\mathrm{Sw}.0 \text{ as } \widehat{\tau}_{\mathrm{reg}})$$

To extract this register, we can simply use the following target expression:

$$\texttt{let in } x_{\mathrm{in1}} = x_{\mathrm{in}}.[15:5]; \texttt{success}$$

The Extract procedure, defined in Algorithm 3, implements this idea by looking for "the right memory path" corresponding to a given high-level path. Its first argument is a description of the form $(x_{\mathrm{in}} \triangleleft p : \tau \text{ as } \widehat{\tau})$ which characterizes the parent/input value from which to extract a subterm. This description indicates that the input location x_{in} contains the memory representation of a (statically

unknown) high-level value of type τ which is known to match the pattern p, according to the memory layout $\widehat{\tau}$. Given such an input description, a destination x_{out} and a high-level path π, it emits code to store in x_{out} the representation of the subterm located at π within the input value. Here, we are looking for "the right memory path", as such, there should exist a fragment within $\widehat{\tau}$ that exactly corresponds to the (high-level) subterm π. Note that we do not specify the desired memory layout for this subterm: for now, we assume that any fragment corresponding to π is acceptable. All we have to do now is to find *where*.

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

```
1  function Extract_Δ((x_in ◁ p : τ as τ̂), x_out, π):
     // Base case: the desired subterm is exactly the input value.
2    if π = ε then
3      return x_out := x_in; success
4    else // Otherwise, explore each branch of the input memory type.
5      B ← for (p_b, τ_b, τ̂_b, frags_b) ∈ Explore_Δ(p, τ, τ̂) do
         // Look for a fragment encoding a parent of the desired subterm.
6        if ∃(π̂ ↦ π_0 as τ̂') ∈ frags_b, ∃π', π = π_0.π' then
7          x'_in ← fresh symbol
8          τ', p' ← focus_Δ (π_0, τ_b) , focus (π_0, p_b)
           // Recursively explore this parent subterm.
9          ẽ ← Extract_Δ((x'_in ◁ p' : τ' as τ̂'), x_out, π')
10         ẽ_b ← let in x'_in = x_in.π̂; ẽ
11       else ẽ_b ← fail
12       yield (p_b, ẽ_b)
       // Dynamically determine the branch used for the actual input value.
13     return Destruct_Δ (x_in, τ̂, B)
```

Algorithm 3: Extract: an incomplete compilation algorithm for variable accessors.

If $\pi = \varepsilon$, then the desired subterm is the input value itself: x_{in} already contains its memory representation, and we simply copy its contents to the destination x_{out} (Line 2). Otherwise, we must *explore* the memory type $\widehat{\tau}$ to find the position of the desired subterm's representation, depending on its actual value. Indeed, a key specificity of the compilation of accessors is that *the precise shape of the accessed value is statically unknown*. While a value v always matches at most one branch of a split memory type, the actual interpretation of an accessor $x.\pi$ depends on the value bound to x, which is not available at compile time. We must therefore emit code which dynamically determines the precise shape of x, and uses this information to find the representation of $x.\pi$. Let us illustrate this on an example.

Example 5.15 (Variable head location in lists, cont'). In Example 5.3, we already showed that the exact position of a given subterm within a value's memory representation was not always fixed. Recall the following type definitions from Example 3.4:

$$\Delta_{\text{list}} = \{t_{\text{list}} \mapsto \tau_{\text{list}}, t_p \mapsto \widehat{\tau}_p\} \qquad \tau_{\text{list}} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{\text{list}}\rangle)$$

$$\widehat{\tau}_p = \text{split}\,(.[0:2])\,\{$$

$$\begin{array}{lll} 0 \text{ from } \text{Nil} & \Rightarrow & \__{64} \ltimes [0:2] : (0)_2 \\ 1 \text{ from } \text{Cons}(\langle \_, \text{Nil}\rangle) & \Rightarrow & \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons.0 as } I_{32}) \\ 2 \text{ from } \text{Cons}(\langle \_, \text{Cons}(\_)\rangle) & \Rightarrow & \&_{64}\left(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1.Cons.0 as } I_{32}), (.\text{Cons.1.Cons.1 as } t_p)\}\!\}\right) \ltimes [0:2] : (2)_2 \end{array}$$

$$\}$$

Consider a call to Extract with the following arguments:

$$\textsc{Extract}_{\Delta_{\text{list}}}\left((x_{\text{in}} \triangleleft \text{Cons}(\langle \_, \_ \rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), x_{\text{out}}, .\text{Cons}.0\right)$$

Here, our goal is to emit code which stores in x_{out} the memory representation of the subterm at position .Cons.0 within the (high-level) input value of type τ_{list}. This input value is statically unknown; however, it is known to match the pattern $\text{Cons}(\langle \_, \_ \rangle)$, and its memory representation as $\widehat{\tau}_p$ is known to be stored in the input location x_{in}.

While this information is enough to exclude the Nil case (in which it would be impossible to extract the subterm at .Cons.0), it is not sufficient to determine the exact layout of the concrete contents of x_{in}. Indeed, there are two different branches in the root split of $\widehat{\tau}_p$ whose provenances are compatible with $\text{Cons}(\langle \_, \_ \rangle)$.

- If the concrete input memory value represents a list with a single element, then its precise layout follows the $\text{Cons}(\langle \_, \text{Nil} \rangle)$ branch, and the fragment $(.\text{Cons}.0 \text{ as } I_{32})$ representing the desired subterm is located at $.[2:32]$.
- Otherwise, the contents of x_{in} follow the $\text{Cons}(\langle \_, \text{Cons}(\_) \rangle)$ branch, in which the fragment $(.\text{Cons}.0 \text{ as } I_{32})$ is located at $.*.0$.

△

To handle such situations, we use the **Explore** function on line 5. **Explore**, defined in Fig. 5.14, combines memory type specialization with **shatter** to examine both splits and fragments. Given a high-level type τ, a memory type $\widehat{\tau}$ which agrees with τ (both considered in the type variable environment Δ) and a pattern p of type τ, $\textbf{Explore}_\Delta(p, \tau, \widehat{\tau})$ yields a list of *branches* of the form $(p_b, \tau_b, \widehat{\tau}_b, \text{frags}_b)$. These branches characterize the space of high-level values matched by the pattern p and of their memory representations. Each branch consists of a precise pattern p_b, the specialized type τ_b and layout $\widehat{\tau}_b$ for this pattern, and the list of fragments and primitives contained therein frags_b.

$$\textbf{Explore}_\Delta(p_0, \tau_0, \widehat{\tau}_0) = \left\{ \left(p, \tau_0 \,/\, p, \widehat{\tau}, \textbf{shatter}_\Delta(\widehat{\tau})\right) \;\middle|\; (p, \widehat{\tau}) \in \widehat{\tau}_0 \,/\, p_0 \right\}$$

Figure 5.14: **Explore** the branches of a given memory layout.

Example 5.16 (**Explore** the packed list layout.)**.** Recall the following Extract call from Example 5.15:

$$\textsc{Extract}_{\Delta_{\text{list}}}\left((x_{\text{in}} \triangleleft \text{Cons}(\langle \_, \_ \rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), x_{\text{out}}, .\text{Cons}.0\right)$$

where

$$\Delta_{\text{list}} = \{t_{\text{list}} \mapsto \tau_{\text{list}}, t_p \mapsto \widehat{\tau}_p\} \qquad \tau_{\text{list}} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{\text{list}} \rangle)$$

$\widehat{\tau}_p = \text{split}\,(.[0:2])\,\{$

$$\begin{array}{lll} 0 \text{ from } \text{Nil} & \Rightarrow & \_{64} \ltimes [0:2] : (0)_2 \\ 1 \text{ from } \text{Cons}(\langle \_, \text{Nil} \rangle) & \Rightarrow & \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons}.0 \text{ as } I_{32}) \\ 2 \text{ from } \text{Cons}(\langle \_, \text{Cons}(\_) \rangle) & \Rightarrow & \&_{64}\left(\left\{\!\left\{(.\text{Cons}.0 \text{ as } I_{32}), (.\text{Cons}.1.\text{Cons}.0 \text{ as } I_{32}), (.\text{Cons}.1.\text{Cons}.1 \text{ as } t_p)\right\}\!\right\}\right) \ltimes [0:2] : (2)_2 \end{array}$$

$\}$

We explore all branches $\widehat{\tau}_p$ which are relevant to the input pattern $\text{Cons}(\langle \_, \_ \rangle)$:

$$\textbf{Explore}_{\Delta_{\text{list}}}\left(\text{Cons}(\langle \_, \_ \rangle), \tau_{\text{list}}, \widehat{\tau}_p\right) = \left\{ \begin{array}{l} \left(\text{Cons}(\langle \_, \text{Nil} \rangle), \text{Cons}(\langle I_{32}, \text{Nil} \rangle), \widehat{\tau}_1, \text{frags}_1\right) \\ \left(\text{Cons}(\langle \_, \text{Cons}(\_) \rangle), \text{Cons}(\langle I_{32}, \text{Cons}(t_{\text{list}}) \rangle), \widehat{\tau}_2, \text{frags}_2\right) \end{array} \right\}$$

where

$$\widehat{\tau}_1 = \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons}.0 \text{ as } I_{32}) \qquad \text{frags}_1 = \{(.[2:32] \mapsto .\text{Cons}.0 \text{ as } I_{32})\}$$

$$\widehat{\tau}_2 = \&_{64}\left(\left\{\!\left\{ \begin{array}{l} (.\text{Cons}.0 \text{ as } I_{32}), \\ (.\text{Cons}.1.\text{Cons}.0 \text{ as } I_{32}), \\ (.\text{Cons}.1.\text{Cons}.1 \text{ as } t_p) \end{array} \right\}\!\right\}\right) \ltimes [0:2] : (2)_2 \qquad \text{frags}_2 = \left\{ \begin{array}{l} (.*.0 \mapsto .\text{Cons}.0 \text{ as } I_{32}), \\ (.*.1 \mapsto .\text{Cons}.1.\text{Cons}.0 \text{ as } I_{32}), \\ (.\text{Cons}.1.\text{Cons}.1 \text{ as } t_p).*.2 \end{array} \right\}$$

Note that the types and layouts of branches are *specialized* according to their patterns $\mathrm{Cons}(\langle \_, \mathrm{Nil} \rangle)$ and $\mathrm{Cons}(\langle \_, \mathrm{Cons}(\_) \rangle)$ respectively. △

After gathering all possible branches with **Explore**, we process each of them by emitting specialized target code which extracts the memory representation of the desired subterm $.\pi$ from the contents of x_{in}. For each branch $(p_b, \tau_b, \widehat{\tau}_b, \text{frags}_b)$, we proceed as follows. The main idea is to look for a part of $\widehat{\tau}_b$ which *represents a parent of the desired subterm*, i.e., a fragment (or primitive) $(\widehat{\pi} \mapsto \pi_0 \text{ as } \widehat{\tau}') \in \text{frags}_b$ such that π_0 is a prefix of π. If it exists, we narrow down our search to this specific part (at position $\widehat{\pi}$) of the input value. We bind this part to a new input location, focus its type and pattern accordingly (line 8), then recursively Extract the desired subterm from this new input value (line 10). If no such part exists, we are unable to determine a memory position containing the representation of the subterm at π: in this case, we emit a `fail` statement.

At this point, we have determined how to extract the desired subterm's representation from x_{in} for every possible branch. We must now assemble these specialized target expressions into a single program. At runtime, this program must determine which of the branches gathered by **Explore** corresponds to the actual, concrete input value. This is precisely what our pattern matching compilation approach was designed for: this task is easily carried out using Destruct on line 13 to emit code which inspects the input value in x_{in}, determines which branch p_b matches it and continues with the associated target expression $\tilde{e}_b$. In examples, we will display the code emitted by Destruct in yellow boxes.

Example 5.17 (Extract the first element of a packed list.)**.** Let us consider again the first element of a non-empty list with the following Extract call:

$$\textsc{Extract}_{\Delta_{\text{list}}}\left((x_{\text{in}} \triangleleft \mathrm{Cons}(\langle \_, \_ \rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), x_{\text{out}}, .\mathrm{Cons}.0\right)$$

Recall the branches collected with **Explore** in Example 5.16:

$$\textbf{Explore}_{\Delta_{\text{list}}}\left(\mathrm{Cons}(\langle \_, \_ \rangle), \tau_{\text{list}}, \widehat{\tau}_p\right) = \left\{ \begin{array}{l} (\mathrm{Cons}(\langle \_, \mathrm{Nil} \rangle), \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Nil} \rangle), \widehat{\tau}_1, \text{frags}_1) \\ (\mathrm{Cons}(\langle \_, \mathrm{Cons}(\_) \rangle), \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Cons}(t_{\text{list}}) \rangle), \widehat{\tau}_2, \text{frags}_2) \end{array} \right\}$$

where

$$\widehat{\tau}_1 = \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}) \qquad \text{frags}_1 = \{(.[2:32] \mapsto .\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})\}$$

$$\widehat{\tau}_2 = \&_{64}\left(\left\{\!\!\left\{ \begin{array}{l} (.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), \\ (.\mathrm{Cons}.1.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), \\ (.\mathrm{Cons}.1.\mathrm{Cons}.1 \text{ as } t_p) \end{array} \right\}\!\!\right\}\right) \ltimes [0:2] : (2)_2 \qquad \text{frags}_2 = \left\{ \begin{array}{l} (.*.0 \mapsto .\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), \\ (.*.1 \mapsto .\mathrm{Cons}.1.\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32}), \\ (.\mathrm{Cons}.1.\mathrm{Cons}.1 \text{ as } t_p).*.2 \end{array} \right\}$$

We now emit specialized target code for both of these branches.

- The first branch $(\mathrm{Cons}(\langle \_, \mathrm{Nil} \rangle), \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Nil} \rangle), \widehat{\tau}_1, \text{frags}_1)$ corresponds to lists containing a single element. The fragment $(.[2:32] \mapsto .\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})$ in frags_1 represents exactly the desired subterm $.\mathrm{Cons}.0$. We bind the location at position $.[2:32]$ within x_{in} to a fresh input location symbol x_{in1}, then recursively explore it with the following base case call:

  $$\textsc{Extract}_{\Delta_{\text{list}}}\left((x_{\text{in1}} \triangleleft \_ : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}), x_{\text{out}}, \varepsilon\right)$$

  We emit the following target expression for this branch:

  $$\tilde{e}_1 = \texttt{let in } x_{\text{in1}} = x_{\text{in}}.[2:32];\ x_{\text{out}} := x_{\text{in1}};\ \texttt{success}$$

- The second branch $(\mathrm{Cons}(\langle \_, \mathrm{Cons}(\_) \rangle), \mathrm{Cons}(\langle \mathrm{I}_{32}, \mathrm{Cons}(\langle \mathrm{I}_{32}, t_{\text{list}} \rangle) \rangle), \widehat{\tau}_2, \text{frags}_2)$ corresponds to lists with more than one element. Similar to the first branch, we focus on the fragment $(.*.0 \mapsto .\mathrm{Cons}.0 \text{ as } \mathrm{I}_{32})$ in frags_2, binding it to the fresh input location x_{in2} and recursively exploring it with the following base case call:

  $$\textsc{Extract}_{\Delta_{\text{list}}}\left((x_{\text{in2}} \triangleleft \_ : \mathrm{I}_{32} \text{ as } \mathrm{I}_{32}), x_{\text{out}}, \varepsilon\right)$$

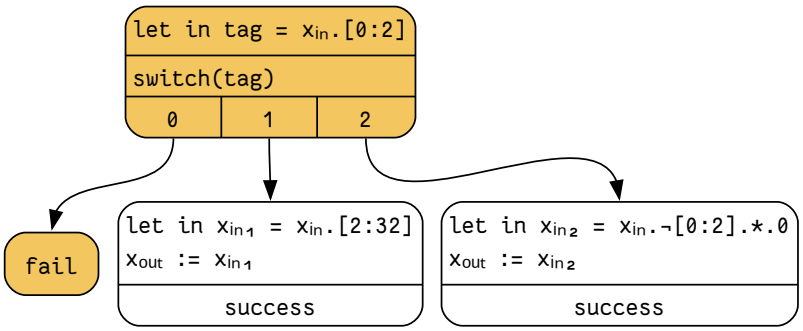
  We emit the following target expression for this branch:

  $$\tilde{e}_2 = \texttt{let in } x_{\text{in2}} = x_{\text{in}}.*.0;\ x_{\text{out}} := x_{\text{in2}};\ \texttt{success}$$

We then emit target code which dynamically selects the appropriate branch $\tilde{e}_1$ or $\tilde{e}_2$ with the following Destruct call:

$$\textsc{Destruct}_{\Delta_{\text{list}}}\left((x_{\text{in}} : \widehat{\tau}_{\text{p}}), \{(\text{Cons}(\langle \_, \text{Nil}\rangle), \tilde{e}_1), (\text{Cons}(\langle \_, \text{Cons}(\_)\rangle), \tilde{e}_2)\}\right)$$

We finally obtain the target program represented by the following control flow graph:



Note that the `fail` branch generated for the 0 tag value, which corresponds to the Nil branch of $\widehat{\tau}_{\text{p}}$, is never taken since x_{in} represents a non-empty list. Going forward, we will omit such unreachable switch branches from CFGs. △

To implement Extract, we assumed that once we have narrowed down the layout $\widehat{\tau}$ to a single branch $\widehat{\tau}_{\text{b}}$, any given high-level path π can be translated to a single memory path $\widehat{\pi}$. While this is true for some configurations, it is not the case in general, as demonstrated by the following example.

Example 5.18. Recall the RISC-V types and values from Example 5.1:

$$\begin{aligned}\tau_{\text{riscv}} = \ & \text{Add}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \tau_{\text{reg}}\rangle)\\ & \mid \text{Addi}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \text{I}_{12}\rangle)\\ & \mid \text{Jal}(\langle \tau_{\text{reg}}, \text{I}_{20}\rangle)\\ & \mid \text{Sw}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \text{I}_{12}\rangle)\end{aligned} \qquad \widehat{\tau}_{\text{riscv}} = \text{split}\,(.[0:7])\left\{\begin{array}{lll}\texttt{0x33} & \text{from Add}(\_) & \Rightarrow \widehat{\tau}_{\text{Add}}\\ \texttt{0x13} & \text{from Addi}(\_) & \Rightarrow \widehat{\tau}_{\text{Addi}}\\ \texttt{0x6f} & \text{from Jal}(\_) & \Rightarrow \widehat{\tau}_{\text{Jal}}\\ \texttt{0x23} & \text{from Sw}(\_) & \Rightarrow \widehat{\tau}_{\text{Sw}}\end{array}\right\}$$

$$\begin{aligned}\widehat{\tau}_{\text{Sw}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x23})_7\\ & \ltimes [7:5] : (.\text{Sw}.2.[0:5] \text{ as } \text{I}_5)\\ & \ltimes [12:3] : (2)_3\\ & \ltimes [15:5] : (.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})\\ & \ltimes [20:5] : (.\text{Sw}.1 \text{ as } \widehat{\tau}_{\text{reg}})\\ & \ltimes [25:7] : (.\text{Sw}.2.[5:7] \text{ as } \text{I}_7)\end{aligned} \qquad \begin{aligned}v = \ & \text{Sw}(\langle X_1, X_2, 42\rangle)\\ \\ \widehat{v} = \ & \__{32} \ltimes [0:7] : (\texttt{0x23})_7 \ltimes [7:5] : (10)_5 \ltimes [12:3] : (2)_3\\ & \ltimes [15:5] : (1)_5 \ltimes [20:5] : (2)_5 \ltimes [25:7] : (1)_7\end{aligned}$$

The subterms at positions .Sw.0 and .Sw.1 within v, which correspond to the two register operands of an Sw instruction, are both accessible as a single fragment within its memory representation $\widehat{v}$, at positions .[15 : 5] and .[20 : 5] respectively. Our Extract procedure is able to emit target code which extracts their representations from the parent memory value $\widehat{v}$: we have

$$\textsc{Extract}\left((x_{\text{in}} \triangleleft \text{Sw}(\_) : \tau_{\text{riscv}} \text{ as } \widehat{\tau}_{\text{riscv}}), (x_{\text{out}} : \tau_{\text{reg}} \text{ as } \widehat{\tau}_{\text{reg}}), \text{Sw}.0\right) = \texttt{let in } x'_{\text{in}} = x_{\text{in}}.[15:5];\ x_{\text{out}} := x'_{\text{in}};\ \texttt{success}$$

However, we are not yet able do the same for the immediate operand at position .Sw.2. Indeed, this subterm is broken into two pieces within $\widehat{\tau}_{\text{Sw}}$: its 7 lowest bits are stored at position .[0 : 7], and its 5 highest bits at .[25 : 7]. Our Extract procedure is therefore unable to emit adequate target code, since there does not exist any fragment within $\widehat{\tau}_{\text{Sw}}$ which covers the entire subterm .Sw.2. △

In this section, we have shown how tools such as Refine and **Explore** can be combined to design fairly complex compilation procedures such as Construct and Extract. However, these procedures are incomplete. On one hand, Construct is by design limited to statically known value constructors. On the other hand, Extract fails to retrieve subterms such as the immediate operand at position .Sw.2 from

a τ_{riscv} value. In order to handle such situations, we must emit target code which assembles multiple pieces together into a single memory value. This is done by our full compilation procedure, which we present in the next section. As we will see, it leverages previous tools to compile any given pivot expression to equivalent target code.

5.5 Compilation of Arbitrary Valuexpressions

We are now ready to present our main compilation approach for pivot expressions $(u : \tau \text{ as } \widehat{\tau})$, which consists of two mutually recursive procedures: Rebuild (Algorithm 5) and Seek (Algorithm 4). Seek is an extended version of Extract which looks for a fragment corresponding to a given high-level subterm. Unlike Extract, if no such fragment exists, it defers to Rebuild which will break down the desired subterm into smaller pieces. Rebuild is somewhat similar to Construct: it handles the constant parts of the desired valuexpression u, using Refine to build a first skeleton and Seek to fill its variable parts. The main ideas underlying both algorithms are:

- Alternatively **Explore** input and output values, using Seek and Rebuild respectively.
- Identify each input value with a quadruple $(x_{in} \triangleleft p : \tau \text{ as } \widehat{\tau})$ and the output value with a triple $(x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out})$.
- Break down the desired valuexpression u until we reach the smallest possible pieces – individual bits of primitive values – which are necessarily directly available within $\widehat{\tau}$ (assuming that $\widehat{\tau}$ is valid, well-kinded and agrees with τ).
- Since the exact layout of data may be statically unknown, we use Destruct to emit code which dynamically determines the precise shape of the output value by inspecting input values, and branches to the corresponding specialized program. As in Extract, we will highlight the code emitted by Destruct in yellow.

5.5.1 Seek a subterm

Seek, in Algorithm 4, compiles an accessor $x.\pi$ to target code. We consider the memory representation of the value bound to x (which is already built and read-only at execution time) as the *input value* and identify it with the quadruple $(x_{in} \triangleleft p_{in} : \tau_{in} \text{ as } \widehat{\tau}_{in})$, composed of its input location, type, layout and pattern respectively. That is, at execution time, x_{in} is expected to contain the memory representation according to $\widehat{\tau}_{in}$ of some value of type τ_{in} matching the pattern p_{in}. Similarly, we refer to the memory representation of the desired subterm (which is the piece of data currently being built) as the *output value* and identify it with the triple $(x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out})$ composed of its output location, type and layout respectively. Our goal is to emit code that, given an input location x_{in} containing the memory representation of v following the layout $\widehat{\tau}_{in}$ and an output location x_{out}, stores in x_{out} the representation of **focus** (π, v) following the layout $\widehat{\tau}_{out}$.

Seek is similar to the Extract procedure presented in Section 5.4.2. The differences are highlighted in orange. As we have seen, in some cases, there exists no single fragment that covers the desired path. Alternatively, we might reach $\pi = \varepsilon$, but $\widehat{\tau}_{in} \neq \widehat{\tau}_{out}$, indicating that we should exhibit an *isomorphism* between $\widehat{\tau}_{in}$ and $\widehat{\tau}_{out}$ to get the desired memory value. In both cases, we need to break down the desired output value into smaller pieces in order to reconstruct it differently, on Line 9. For primitive types (Line 10), we expose the underlying individual bits using a `cast` operation to consider the output value as a composite word. We restore its primitive integer type after rebuilding its contents with another `cast` [1]. For other memory types, these smaller pieces correspond to fragments and primitives. We now need to *rebuild* the output value from these pieces. This task is carried out by our next procedure: Rebuild.

[1] While correct, rebuilding each bit individually is not optimal. Ideally, we would explore the input values to determine which pieces are available and avoid unnecessarily breaking down values.

Data: Δ the type variable environment
Data: $(x_{in} \triangleleft p_{in} : \tau_{in}$ as $\widehat{\tau}_{in})$ the input description
Data: $(x_{out} : \tau_{out}$ as $\widehat{\tau}_{out})$ the output description
Data: π the path to the desired subterm within the input value
Result: Code binding x_{out} to the representation of the subterm at position π in the input value

function $\textsc{Seek}_\Delta((x_{in} \triangleleft p_{in} : \tau_{in}$ as $\widehat{\tau}_{in}), (x_{out} : \tau_{out}$ as $\widehat{\tau}_{out}), \pi)$**:**
  // Invariant: π and p_{in} are compatible.
  // Base case: input and output values are the same data with the same representation.
  if $\pi = \varepsilon \wedge \widehat{\tau}_{in} = \widehat{\tau}_{out}$ **then return** $x_{out} := x_{in}$; `success`
  else // Otherwise, Explore all cases of the input value.
    $B \leftarrow$ **for** $(p_b, \tau_b, \widehat{\tau}_b, frags_b) \in$ **Explore**$_\Delta(p_{in}, \tau_{in}, \widehat{\tau}_{in})$ **do**
      // Seek a fragment containing the piece of data at π.
      if $\exists(\widehat{\pi}_f \mapsto \pi_f$ as $\widehat{\tau}_f) \in frags_b, \exists\pi', \pi = \pi_f.\pi'$ **then**
        // Found one. We focus on it and Seek inside.
        $x_f \leftarrow$ fresh symbol
        $\tau_f, p_f \leftarrow$ **focus** (π_f, τ_b) , **focus** (π_f, p_b)
        $\tilde{e}_b \leftarrow$ `let in` $x_f = x_{in}.\widehat{\pi}_f$; $\textsc{Seek}_\Delta\left((x_f \triangleleft p_f : \tau_f$ as $\widehat{\tau}_f), (x_{out} : \tau_{out}$ as $\widehat{\tau}_{out}), \pi'\right)$
      else // Otherwise, Rebuild from smaller pieces.
        if $\widehat{\tau}_{out} = I_\ell$ **then** // If we are seeking a primitive, decompose it in individual bits
          $\widehat{\tau}'_{out} \leftarrow \__\ell \bowtie_{0 \leqslant i < \ell} [i : 1] : (.[i : 1]$ as $I_1)$
          $\tilde{e} \leftarrow \textsc{Rebuild}\left(\{(x_{in} \triangleleft p_b : \tau_b$ as $\widehat{\tau}_b)\}, (x_{out} : \tau_{out}$ as $\widehat{\tau}'_{out}), x_{in}.\pi\right)$
          $\tilde{p} \leftarrow \__\ell \bowtie_{0 \leqslant i < \ell} [i : 1] : \__1$
          $\tilde{e}_b \leftarrow$ `cast` x_{out} `to` $\tilde{p}$; $\tilde{e}$; `cast` x_{out} `to` I_ℓ; `success`
        else
          $\tilde{e}_b \leftarrow \textsc{Rebuild}\left(\{(x_{in} \triangleleft p_b : \tau_b$ as $\widehat{\tau}_b)\}, (x_{out} : \tau_{out}$ as $\widehat{\tau}_{out}), x_{in}.\pi\right)$
      yield $(p_b, \tilde{e}_b)$
  // Assemble the code of these branches via a decision tree.
  return $\textsc{Destruct}_\Delta(x_{out}, \widehat{\tau}_{in}, B)$

Algorithm 4: Seek – Seek the memory location representing π in x_{in}.

5.5.2 Rebuild a valuexpression

Data: Δ the type variable environment
Data: $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)$ the n input descriptions
Data: $(x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out})$ the output description
Data: u a valuexpression to compile
Result: Target code building the memory representation of u as $\widehat{\tau}_{out}$ in x_{out} from the n input values.

1 **function** REBUILD$_\Delta$ $(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), u)$**:**
  // Base case: target value is a constant encoded as a primitive type.
2 **if** $u = c \wedge \widehat{\tau}_{out} = I_\ell$ **then return** $x_{out} := c$; *success*
3 **else** // Otherwise, explore all cases of the output value.
4 $p_{out0} \leftarrow u[x.\pi \mapsto \_]$
5 $Pos_{out} \leftarrow \{(x.\pi, \pi') \mid \textbf{focus}\ (\pi', u) = x.\pi\}$
6 $p_{out} \leftarrow \textbf{Remap}\ (\{x_i \mapsto p_i \mid 0 \leqslant i < n\}, p_{out0}, Pos_{out})$
7 $B \leftarrow \textbf{for}\ p_b, \tau_b, \widehat{\tau}_b, \mathit{frags}_b \in \textbf{Explore}_\Delta(p_{out}, \tau_{out}, \widehat{\tau}_{out})\ \textbf{do}$
    // Allocate memory, cast and fill in constant parts as needed for this memory type.
8 $\tilde{e}_{const} \leftarrow$ REFINE $(x_{out}, \_|\widehat{\tau}_{out}|, \textbf{shape\_of}_\Delta(\widehat{\tau}_b))$
    // Rebuild target fragments from input values, which we specialize for the current branch.
9 **for** $i \in \{0, \ldots, n-1\}$ **do**
10 $Pos_i \leftarrow \{(x_{out}.\pi, \pi') \mid \textbf{focus}\ (\pi, u) = x_i.\pi'\}$
11 $p'_i \leftarrow \textbf{Remap}(\{x_{out} \mapsto p_b\}, p_i, Pos_i)$
12 $\tilde{e}_{frags} \leftarrow \textbf{for}\ (\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathit{frags}_b\ \textbf{do}$
13 $x_f \leftarrow$ fresh symbol
14 $\tau_f \leftarrow \textbf{focus}\ (\pi_f, \tau_b)$
15 **if** $\exists i \in \{0, \ldots, n-1\}, \exists (x_i.\pi_{in}, \pi_{out}) \in \mathit{Pos}_{out}, \exists \pi, \pi_{out}.\pi = \pi_f$ **then**
        // If this fragment corresponds to a single piece of an input value, Seek it within this value.
16 $\tilde{e}_f \leftarrow$ SEEK$_\Delta((x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i), (x_f : \tau_f \text{ as } \widehat{\tau}_f), \pi_{in}.\pi)$
17 **else** // Otherwise, Rebuild it from smaller pieces.
18 $\tilde{e}_f \leftarrow$ REBUILD$_\Delta(\{(x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_f : \tau_f \text{ as } \widehat{\tau}_f), \textbf{focus}\ (\pi_f, u))$
19 **yield** *let out* $x_f = x_{out}.\widehat{\pi}_f; \tilde{e}_f;$ *freeze*(x_f); *success*
20 **yield** $(\{x_i \mapsto p'_i \mid 0 \leqslant i < n\}, \tilde{e}_{consts}; \tilde{e}_{frags})$
  // Assemble these branches into a decision tree.
21 **return** DESTRUCT$_\Delta$ $(\{x_i : \widehat{\tau}_i \mid 0 \leqslant i < n\}, B)$

Algorithm 5: REBUILD – Rebuild the value in memory representing u from the x_is.

REBUILD (Algorithm 5) compiles an arbitrary valuexpression u to target code that constructs its memory representation as $\widehat{\tau}_{out}$ in the destination x_{out}. Whereas SEEK retrieves a (possibly mangled) relevant fragment from a single input value, REBUILD inspects the shape of the output value to assemble its constituent pieces, which include constant parts of the memory layout as well as fragments extracted from multiple input values. It reuses the general structure of SEEK (and EXTRACT), but considers n input values described by quadruples $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)$ with $0 \leqslant i < n$.

Firstly, when u is a primitive value c and the output layout $\widehat{\tau}_{out}$ is a fixed-width primitive encoding I_ℓ, we simply write the constant c to the output location x_{out} (Line 2). Otherwise, we operate in a similar manner as in the previous procedures EXTRACT and SEEK: we **Explore** all suitable branches of $\widehat{\tau}_{out}$ (Line 3), whose code we will eventually combine with DESTRUCT (Line 21).

However, unlike EXTRACT and SEEK which explore and destruct an already-built input value, REBUILD explores the *output* value. By definition, we cannot use DESTRUCT to determine the shape of the output value, since it is not built yet! Instead, we must infer it from the shapes of the n input values $x_0, \ldots, x_{n-1}$. To do so, we build a pattern p_{out} which is known to match the output value with the following steps.

- On Line 4, we substitute accessors in u with wildcards to get a first approximation of its shape

p_{out0} which matches its statically known parts.

- On Line 5, we capture the variable parts of u by collecting all accessors $x_i.\pi$ that appear in it, along with their positions.
- On Line 6, we finally get a pattern p_{out} which integrates both static and variable information, using the **Remap** function defined in Fig. 5.15.

$$\textbf{Remap}\,(\{x_i \mapsto p_i \mid 0 \leqslant i < n\}, p, \text{Pos}) = p\left[\ .\pi' \leftarrow \textbf{focus}\,(\pi, p_i)\ \mapsto \middle|\ (x_i.\pi, \pi') \in \text{Pos}\ \right]$$

Figure 5.15: **Remap**: refine patterns according to a position map.

The **Remap** function, defined in Fig. 5.15, takes as inputs a map P from variable symbols to patterns, a "parent" pattern p and a set Pos of pairs $(x.\pi, \pi')$. It returns a pattern based on p in which for each pair $(x.\pi, \pi')$, the subterm at position π' has been replaced with **focus** $(\pi, P(x))$.

Example 5.19 (**Remap** list patterns.)**.** Recall $\Delta_{list} = \{t_{list} \mapsto \tau_{list}\}$ and $\tau_{list} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{list}\rangle)$ from Example 5.15. Consider the valuexpression $u = \text{Cons}(\langle x_1.\text{Cons}.0, x_2.\text{Cons}.1\rangle)$ which, given two non-empty lists x_1 and x_2, returns a new list consisting of the head of x_1 appended to the tail of x_2. Its "static" pattern is $\text{Cons}(\langle \_, \_\rangle)$, and we collect its accessors and their positions in the following set:

$$\text{Pos}_{out} = \{(x_1.\text{Cons}.0, .\text{Cons}.0), (x_2.\text{Cons}.1, .\text{Cons}.1)\}$$

We can now use **Remap** to get a precise pattern for u:

$$\textbf{Remap}\,(\{x_1 \mapsto \text{Cons}(\_), x_2 \mapsto \text{Cons}(\_)\}, \text{Cons}(\langle \_, \_\rangle), \text{Pos}_{out}) = \text{Cons}(\langle \_, \_\rangle)$$

△

Using this function, we are able to characterize the output value with a pattern p_{out}, which we use to **Explore** all possible branches of its memory layout $\widehat{\tau}_{out}$ (Line 3). Note that there is a bijection between the p_{out} branches and the possible combinations of p_i branches since, by definition, the only variable parts of u are its accessors $x_i.\pi$.

For each of these branches, we proceed similarly to Construct: we first Refine the contents of the output location x_{out} to capture all constant parts of the memory layout (Line 8). We then move on to the variable parts of the output value. On Line 11, we derive a new pattern p'_i for each input value, which is at least as precise as p_i and also integrates information from the considered output branch p_b. We then build the memory representation of each fragment appearing in $\widehat{\tau}_b$. Depending on whether a given fragment corresponds to a variable accessor in u, we either use Seek or Rebuild to recursively build it in its adequate location. On Line 15, we determine whether the high-level path π_f corresponds to a variable accessor $x_i.\pi$ using the previously computed map Pos_{out}. If so, we can simply Seek the corresponding subterm within the input value x_i (and discard all other input values). Otherwise, we must recursively Rebuild this part of u. As with Construct, in upcoming examples, we will highlight the code emitted by Refine in green and the code emitted to process each fragment in light blue.

On Line 21, we finally assemble each branch into a single target program using Destruct. As mentioned earlier, at execution time, the only way to determine the shape of the desired output value is to infer it from input values. Consequently, we pass the n input values $x_0, \ldots, x_{n-1}$ to Destruct.

Example 5.20 (Rebuild the head of a packed list)**.** Recall the following type definitions from Example 5.15:

$$\Delta_{list} = \{t_{list} \mapsto \tau_{list}, t_p \mapsto \widehat{\tau}_p\} \qquad \tau_{list} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{list}\rangle)$$

$\widehat{\tau}_p = \text{split}\,(.[0:2])\,\{$

$$\begin{array}{lll} 0 \text{ from } \text{Nil} & \Rightarrow & \_{64} \ltimes [0:2] : (0)_2 \\ 1 \text{ from } \text{Cons}(\langle \_, \text{Nil}\rangle) & \Rightarrow & \_{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons}.0 \text{ as } I_{32}) \\ 2 \text{ from } \text{Cons}(\langle \_, \text{Cons}(\_)\rangle) & \Rightarrow & \&_{64}\,(\{\!\{(.\text{Cons}.0 \text{ as } I_{32}), (.\text{Cons}.1.\text{Cons}.0 \text{ as } I_{32}), (.\text{Cons}.1.\text{Cons}.1 \text{ as } t_p)\}\!\}) \\ & & \ltimes [0:2] : (2)_2 \end{array}$$

$\}$

Recall the **Explore**d branches for the pattern Cons(_) from Example 5.16:

$$\textbf{Explore}_{\Delta_{\text{list}}}\left(\text{Cons}(\langle \_, \_\rangle), \tau_{\text{list}}, \widehat{\tau}_{\text{p}}\right) = \left\{ \begin{array}{l} \left(\text{Cons}(\langle \_, \text{Nil}\rangle), \text{Cons}(\langle \text{I}_{32}, \text{Nil}\rangle), \widehat{\tau}_1, \text{frags}_1\right) \\ \left(\text{Cons}(\langle \_, \text{Cons}(\_)\rangle), \text{Cons}(\langle \text{I}_{32}, \text{Cons}(\text{t}_{\text{list}})\rangle), \widehat{\tau}_2, \text{frags}_2\right) \end{array} \right\}$$

where

$$\widehat{\tau}_1 = \__{64} \ltimes [0:2) : (1)_2 \ltimes [2:32] : (.\text{Cons.0 as } \text{I}_{32}) \qquad \text{frags}_1 = \{(.[2:32] \mapsto .\text{Cons.0 as } \text{I}_{32})\}$$

$$\widehat{\tau}_2 = \&_{64}\left(\left\{\!\!\left\{ \begin{array}{l} (.\text{Cons.0 as } \text{I}_{32}), \\ (.\text{Cons.1.Cons.0 as } \text{I}_{32}), \\ (.\text{Cons.1.Cons.1 as } \text{t}_{\text{p}}) \end{array} \right\}\!\!\right\}\right) \ltimes [0:2) : (2)_2 \qquad \text{frags}_2 = \left\{ \begin{array}{l} (.*.0 \mapsto .\text{Cons.0 as } \text{I}_{32}), \\ (.*.1 \mapsto .\text{Cons.1.Cons.0 as } \text{I}_{32}), \\ (.\text{Cons.1.Cons.1 as } \text{t}_{\text{p}}).*.2 \end{array} \right\}$$

Consider the following pivot expression which, given a non-empty list x, rebuilds a list containing only its head.

$$(\text{Cons}(\langle \text{x.Cons.0}, \text{Nil}\rangle) : \tau_{\text{list}} \text{ as } \widehat{\tau}_{\text{p}})$$

We now compile it using our Rebuild and Seek procedures. In this situation, we statically know the exact shape of the output value, since the pattern Cons(⟨_, Nil⟩) corresponds to exactly one branch of the memory layout $\widehat{\tau}_{\text{p}}$. However, we do not know the precise shape of the input value x: its statically known pattern Cons(_) matches two branches of its memory layout $\widehat{\tau}_{\text{p}}$. Consequently, at some point during compilation, we will explore two branches of a Seek call, which we will then join using Destruct.

Here, we detail this compilation process using intermediate results from Examples 5.16, 5.17 and 5.19. The emitted code is shown in Fig. 5.16. We start with the following Rebuild call:

$$\text{Rebuild}_{\Delta_{\text{list}}}\left(\left\{(\text{x} \triangleleft \text{Cons}(\_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_{\text{p}})\right\}, (\text{x}_{\text{out}} : \tau_{\text{list}} \text{ as } \widehat{\tau}_{\text{p}}), \text{Cons}(\langle \text{x.Cons.0}, \text{Nil}\rangle)\right)$$

We explore the output memory layout with the pattern Cons(⟨_, Nil⟩) with **Explore** (Cons(⟨_, Nil⟩), τ_{list}, $\widehat{\tau}_{\text{list}}$), which yields a single branch $\left(\text{Cons}(\langle \_, \text{Nil}\rangle), \text{Cons}(\langle \text{I}_{32}, \text{Nil}\rangle), \widehat{\tau}_1, \text{frags}_1\right)$. We first process the constant parts of the specialized memory type $\widehat{\tau}_1$ using Refine:

$$\tilde{e}_{\text{const}} = \text{Refine}\,(\text{x}_{\text{out}}, \__{64}, \__{64} \ltimes [0:2) : (1)_2 \ltimes [2:32] : \__{32})$$

The next step deals with the only variable part of the output value, which is the head of the list represented by the triple (.[2 : 32] ↦ .Cons.0 as I_{32}) in frags_1. It corresponds to the accessor x.Cons.0 in the considered valuexpression. We bind its destination to a new output location $\texttt{head}_{\text{out}}$, then Seek this piece of data with:

$$\tilde{e} = \text{Seek}_{\Delta_{\text{list}}}\left((\text{x} \triangleleft \text{Cons}(\_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_{\text{p}}), (\texttt{head}_{\text{out}} : \text{I}_{32} \text{ as } \text{I}_{32}), .\text{Cons.0}\right)$$

The input pattern Cons(_) is compatible with two branches of the input memory type $\widehat{\tau}_{\text{p}}$, corresponding to the refined patterns Cons(⟨_, Nil⟩) and Cons(⟨_, Cons(_)⟩). We explore both of these branches:

Branch $\left(\textbf{Cons}(\langle \_, \textbf{Nil}\rangle), \textbf{Cons}(\langle \text{I}_{32}, \textbf{Nil}\rangle), \widehat{\tau}_1, \textbf{frags}_1\right)$**:** We find the fragment (.[2 : 32] ↦ .Cons.0 as I_{32}) in frags_1 and emit a target expression $\tilde{e}_1$ which binds its location to a new symbol `head`, then recursively seeks the desired subterm with:

$$\text{Seek}_{\Delta_{\text{list}}}\left((\texttt{head} \triangleleft \_ : \text{I}_{32} \text{ as } \text{I}_{32}), (\texttt{head}_{\text{out}} : \text{I}_{32} \text{ as } \text{I}_{32}), \varepsilon\right)$$

This Seek call is a base case: the input location `head` contains exactly the desired piece of data, and we write its contents to the destination $\texttt{head}_{\text{out}}$.

Branch $\left(\textbf{Cons}(\langle \_, \textbf{Cons}(\_)\rangle), \textbf{Cons}(\langle \text{I}_{32}, \textbf{Cons}(\langle \text{I}_{32}, \text{t}_{\textbf{list}}\rangle)\rangle), \widehat{\tau}_2, \textbf{frags}_2\right)$**:** We find the fragment (.¬[0 : 2].* .0 ↦ .Cons.0 as I_{32}) in frags_2 and emit a target expression $\tilde{e}_2$ which binds its location to a new symbol `head`, then recursively seeks the desired subterm with the same Seek call as the previous branch:

$$\text{Seek}_{\Delta_{\text{list}}}\left((\texttt{head} \triangleleft \_ : \text{I}_{32} \text{ as } \text{I}_{32}), (\texttt{head}_{\text{out}} : \text{I}_{32} \text{ as } \text{I}_{32}), \varepsilon\right)$$

The Seek call returns a target expression $\tilde{e}$ which determines which branch matches the input value, emitted by the following Destruct call:

$$\textsc{Destruct}_{\Delta_{\text{list}}}\left((x : \widehat{\tau}_p), \{(\text{Cons}(\langle \_, \text{Nil}\rangle), \tilde{e}_1), (\text{Cons}(\langle \_, \text{Cons}(\_)\rangle), \tilde{e}_2)\}\right)$$

We append a `freeze` instruction to $\tilde{e}$ and get a target expression $\tilde{e}_{\text{frags}}$ which processes all variable parts of the output value. Finally, we assemble and return the complete target program $\tilde{e}_{\text{consts}}; \tilde{e}_{\text{frags}}$.

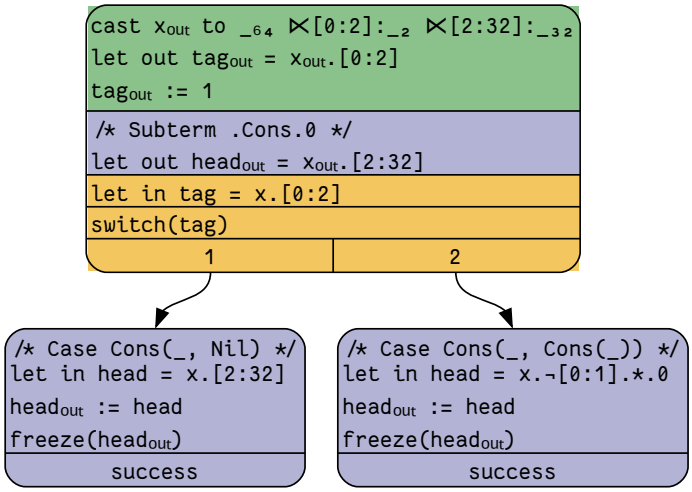


Figure 5.16: Target code emitted by $\textsc{Rebuild}_{\Delta_{\text{list}}}\left(\{(x \triangleleft \text{Cons}(\_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p)\}, (x_{\text{out}} : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), \text{Cons}(\langle x.\text{Cons}.0, \text{Nil}\rangle)\right)$.

△

Together, our Seek and Rebuild procedures are able to handle a variety of situations, including memory types in which integer values are shattered in multiple pieces scattered accross memory such as RISC-V instructions.

Example 5.21. Recall the RISC-V types defined in Example 5.1:

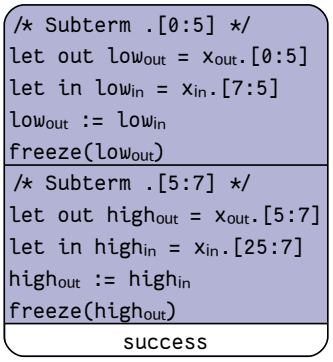
$$\begin{aligned}\tau_{\text{riscv}} = \ & \text{Add}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, \tau_{\text{reg}}\rangle)\\ & \mid \text{Addi}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}\rangle)\\ & \mid \text{Jal}(\langle \tau_{\text{reg}}, I_{20}\rangle)\\ & \mid \text{Sw}(\langle \tau_{\text{reg}}, \tau_{\text{reg}}, I_{12}\rangle)\end{aligned} \qquad \widehat{\tau}_{\text{riscv}} = \text{split}\,(.[0:7]) \left\{ \begin{array}{lll} \texttt{0x33} \text{ from } \text{Add}(\_) & \Rightarrow & \widehat{\tau}_{\text{Add}} \\ \texttt{0x13} \text{ from } \text{Addi}(\_) & \Rightarrow & \widehat{\tau}_{\text{Addi}} \\ \texttt{0x6f} \text{ from } \text{Jal}(\_) & \Rightarrow & \widehat{\tau}_{\text{Jal}} \\ \texttt{0x23} \text{ from } \text{Sw}(\_) & \Rightarrow & \widehat{\tau}_{\text{Sw}} \end{array} \right\}$$

$$\begin{aligned}\widehat{\tau}_{\text{Sw}} = \ & \__{32} \ltimes [0:7] : (\texttt{0x23})_7\\ & \ltimes [7:5] : (.\text{Sw}.2.[0:5] \text{ as } I_5)\\ & \ltimes [12:3] : (2)_3\\ & \ltimes [15:5] : (.\text{Sw}.0 \text{ as } \widehat{\tau}_{\text{reg}})\\ & \ltimes [20:5] : (.\text{Sw}.1 \text{ as } \widehat{\tau}_{\text{reg}})\\ & \ltimes [25:7] : (.\text{Sw}.2.[5:7] \text{ as } I_7)\end{aligned}$$

In Example 5.18, we showed that Extract was unable to retrieve the immediate operand of a store-word RISC-V instruction. Using Rebuild and Seek, we are now able to do so by collecting the two relevant pieces of data and reassembling them into a single I_{32} value. We achieve this with the following Seek call:

$$\textsc{Seek}\left((x_{\text{in}} \triangleleft \text{Sw}(\langle \_, \_, \_\rangle) : \tau_{\text{riscv}} \text{ as } \widehat{\tau}_{\text{riscv}}), (x_{\text{out}} : I_{32} \text{ as } I_{32}), .\text{Sw}.2\right)$$

which yields the following target program (note that we have chosen to break down the output value "optimally", rather than in individual bits as specified in the general Seek algorithm):



△

In this section, we described two procedures Seek and Rebuild which, together with Destruct, provide all of the necessary tools to compile the full space of source expressions. In the next section, we will show how to ensure termination of these algorithms in the presence of recursive combinations of memory layouts, and finally provide a unified compilation procedure for the ribbit language.

5.6 Wrapping up: a complete compilation procedure for Ribbit

5.6.1 Dealing with recursion

Although the compilation approach presented in the previous section is sufficient to handle most situations, it does not necessarily terminate in the presence of recursive types and layouts. Let us demonstrate this on an example.

Example 5.22 (Recursive rebuilding of linked lists)**.** Recall the following type definitions from Example 5.20:

$$\Delta_{\text{list}} = \{t_{\text{list}} \mapsto \tau_{\text{list}}, t_p \mapsto \widehat{\tau}_p, t_c \mapsto \widehat{\tau}_c\} \qquad \tau_{\text{list}} = \text{Nil} \mid \text{Cons}(\langle I_{32}, t_{\text{list}}\rangle)$$

$$\widehat{\tau}_p = \text{split}\,(.[0:2])\,\{$$

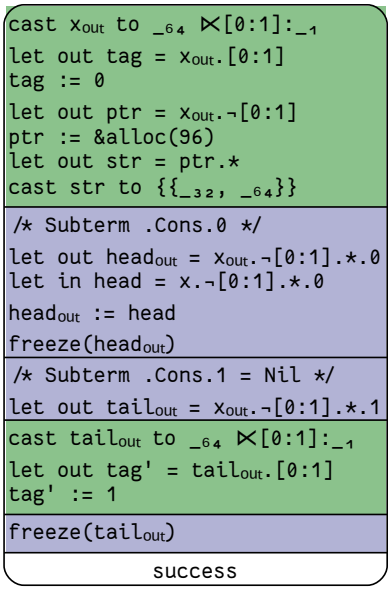
$$\begin{array}{lll} 0 \text{ from } \text{Nil} & \Rightarrow & \__{64} \ltimes [0:2] : (0)_2 \\ 1 \text{ from } \text{Cons}(\langle \_, \text{Nil}\rangle) & \Rightarrow & \__{64} \ltimes [0:2] : (1)_2 \ltimes [2:32] : (.\text{Cons.0 as } I_{32}) \\ 2 \text{ from } \text{Cons}(\langle \_, \text{Cons}(\_)\rangle) & \Rightarrow & \&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1.Cons.0 as } I_{32}), (.\text{Cons.1.Cons.1 as } t_p)\}\!\}) \\ & & \ltimes [0:2] : (2)_2 \end{array}$$

$$\}$$

Consider the simply-linked list layout $\widehat{\tau}_c$, which we introduced in Example 3.4:

$$\widehat{\tau}_c = \text{split}\,(.[0:1])\left\{\begin{array}{lll} 1 \text{ from } \text{Nil} & \Rightarrow & \__{64} \ltimes [0:1] : (1)_1 \\ 0 \text{ from } \text{Cons}(\_) & \Rightarrow & \&_{64}\,(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1 as } t_c)\}\!\}) \ltimes [0:1] : (0)_1 \end{array}\right\}$$

On its own, this naive memory layout is easily handled with our existing compilation procedures Rebuild, Seek and Destruct. The two memory layouts $\widehat{\tau}_c$ and $\widehat{\tau}_p$ specify two different ways to encode the same inductive structure (lists). For instance, consider the valuexpression $u = \text{Cons}(\langle x.\text{Cons.0}, \text{Nil}\rangle)$ where x is bound to a τ_{list} value. As we have seen in Example 5.20, we can compile the pivot expression $(u : \tau_{\text{list}} \text{ as } \widehat{\tau}_p)$ (assuming that x is represented as $\widehat{\tau}_p$) with $\textsc{Rebuild}_{\Delta_{\text{list}}}\,((x \triangleleft \text{Cons}(\_) : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), (x_{\text{out}} : \tau_{\text{list}} \text{ as } \widehat{\tau}_p), u)$. Similarly, if x is represented as $\widehat{\tau}_c$, we can compile the pivot expression $(u : \tau_{\text{list}} \text{ as } \widehat{\tau}_c)$ with $\textsc{Rebuild}_{\Delta_{\text{list}}}\,((x \triangleleft \text{Cons}(\_) : \tau_{\text{list}} \text{ as}$ which yields the following target program:



However, using both of these layouts together raises new issues. For instance, given a τ_{list} value whose memory representation using the simple layout $\widehat{\tau}_c$ is stored in x, consider the following pivot expression, which reencodes it using the packed layout $\widehat{\tau}_p$: $(x.\varepsilon : \tau_{list}$ as $\widehat{\tau}_p)$. We can readily ask Seek to emit such conversion code:

$$\textsc{Seek}_{\Delta_{list}}\left((x \triangleleft \_ : t_{list} \text{ as } t_c), (x_{out} : t_{list} \text{ as } t_p), \varepsilon\right)$$

Nevertheless, emitting code which transforms an entire inductive data structure in memory is not so simple: it requires *recursive* target code to walk the entire list structure and fuse blocks two-by-two, as shown in Fig. 5.17. Given this input, our current algorithms will not terminate: we will attempt to Rebuild/Seek each element of the list without ever converging. Let us demonstrate this by walking through each compilation step of the above Seek call.

We first **Explore** the input memory layout $\widehat{\tau}_c$. We get two branches corresponding to the patterns Nil and Cons(_), for which we will emit two target expressions denoted $\tilde{e}_{Nil}$ and $\tilde{e}_{Cons}$ respectively. We emit code which determines which branch matches the input value with the following Destruct call:

$$\textsc{Destruct}_{\Delta_{list}}\left((x : t_c), \{(Nil, \tilde{e}_{Nil}), (Cons(\_), \tilde{e}_{Cons})\}\right)$$

which yields a switch node inspecting the split discriminant position $.[0:1]$.

Neither of these two branches contains any fragment representing a parent of the desired subterm ε. We must therefore rebuild it from smaller pieces with the following Rebuild calls:

$$\tilde{e}_{Nil} = \textsc{Rebuild}_{\Delta_{list}}\left(\{(x \triangleleft Nil : Nil \text{ as } \_{64} \ltimes [0:1] : (1)_1)\}, (x_{out} : t_{list} \text{ as } t_p), x.\varepsilon\right)$$

$$\tilde{e}_{Cons} = \textsc{Rebuild}_{\Delta_{list}}\left(\{(x \triangleleft Cons(\_) : Cons(\langle I_{32}, t_{list}\rangle) \text{ as } \&_{64}(\dots) \ltimes [0:1] : (0)_1)\}, (x_{out} : t_{list} \text{ as } t_p), x.\varepsilon\right)$$

Following the Rebuild algorithm, we first determine which pattern is known to match the output value using its specified valuexpression and each input value's pattern. Here, the desired output value encodes exactly the same data as the input value x (as specified by the accessor $x.\varepsilon$). We then **Explore** all branches of the output memory type $\widehat{\tau}_p$ which are compatible with this pattern. Following the three-branch split in $\widehat{\tau}_p$, we get a single branch in the Nil case and two branches $Cons(\langle \_, Nil\rangle)$ and $Cons(\langle \_, Cons(\_)\rangle)$ in the Cons(_) case. To distinguish between the two possible branches in $\tilde{e}_{Cons}$, we use the following Destruct call, where $\tilde{e}_{ConsNil}$ and $\tilde{e}_{ConsCons}$ designate the target code emitted for each branch:

$$\textsc{Destruct}_{\Delta_{list}}\left((x : \widehat{\tau}_c), \{(Cons(\langle \_, Nil\rangle), \tilde{e}_{ConsNil}), (Cons(\langle \_, Cons(\_)\rangle), \tilde{e}_{ConsCons})\}\right)$$

which yields a switch node inspecting the nested discriminant at position $.\neg[0:1].*.1.[0:1]$ within x.

Let us now detail the code emitted for each of these three branches $\tilde{e}_{\text{Nil}}$, $\tilde{e}_{\text{ConsNil}}$ and $\tilde{e}_{\text{ConsCons}}$. Following the Rebuild algorithm, we first use Refine to emit code which allocates, casts and initializes memory in x_{out} according to the shape of the considered branch's memory type. Here, all three branches cast the contents of x_{out} to a composite word shape revealing the $\widehat{\tau}_p$ split discriminant location $.[0:2]$, whose contents are then initialized to the appropriate discriminant value. In the $\text{Cons}(\langle \_, \text{Cons}(\_)\rangle)$ case, we also allocate 128 bits of memory to store a struct with two 32-bit and one 64-bit fields.

For the empty list Nil, this is sufficient and the program ends with `success`. For the two remaining branches $\text{Cons}(\langle \_, \text{Nil}\rangle)$ and $\text{Cons}(\langle \_, \text{Cons}(\_)\rangle)$, the next step is to fill the location of each fragment in the output memory type with its concrete contents. For the $\text{Cons}(\langle \_, \text{Nil}\rangle)$ branch, we must retrieve the single element of the list x.Cons.0 and store it at position $.[2:32]$ within x_{out}. We bind its destination to a new output location head_{out} and perform the following Seek call:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((x \triangleleft \text{Cons}(\langle \_, \text{Nil}\rangle)) : \text{Cons}(\langle I_{32}, \text{Nil}\rangle) \text{ as } \&_{64}(\dots) \ltimes [0:1] : (0)_1), (\text{head}_{\text{out}} : I_{32} \text{ as } I_{32}), .\text{Cons.0}\right)$$

Its memory representation as I_{32} is stored at position $.*.0$ within the input location x. We bind this location to a new symbol head and recursively Seek the desired subterm from it:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((\text{head} \triangleleft \_ : I_{32} \text{ as } I_{32}), (\text{head}_{\text{out}} : I_{32} \text{ as } I_{32}), \varepsilon\right)$$

We have reached the base case of Seek and extract the subterm by copying the contents of head into its destination head_{out}.

In the $\text{Cons}(\langle \_, \text{Cons}(\_)\rangle)$ case, we proceed similarly for the first and second elements of the list x.Cons.0 and x.Cons.1.Cons.0. Let us now focus on the last subterm of this branch, which is the tail of the list x.Cons.1.Cons.1. We first bind its destination, which is at position $.*.2$ within the root output location x_{out}, to a new output location tail_{out}. We then Seek it within x with the following call:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((x \triangleleft \text{Cons}(\langle \_, \text{Cons}(\_)\rangle)) : \text{Cons}(\langle I_{32}, \text{Cons}(\dots)\rangle) \text{ as } \&_{64}(\dots) \ltimes [0:1] : (0)_1), (\text{tail}_{\text{out}} : t_{\text{list}} \text{ as } t_p), .\text{Cons.1.Cons.1}\right)$$

The input memory type has already been specialized from $\widehat{\tau}_c$ to the following split-free type:

$$\&_{64}(\{\!\{(.\text{Cons.0 as } I_{32}), (.\text{Cons.1 as } t_c)\}\!\}) \ltimes [0:1] : (0)_1$$

It contains the fragment $(.\text{Cons.1 as } t_c)$, which represents a parent of the desired subterm .Cons.1.Cons.1. We bind its location in memory, which is at position $.\neg[0:1].*.1$ within x, to a new symbol tail, and recursively seek the desired subterm in it with:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((\text{tail} \triangleleft \text{Cons}(\_) : \text{Cons}(\langle I_{32}, t_{\text{list}}\rangle) \text{ as } t_c), (\text{tail}_{\text{out}} : t_{\text{list}} \text{ as } t_p), .\text{Cons.1}\right)$$

We explore the only branch of $\widehat{\tau}_c$ which is compatible with the input pattern Cons(_), which yields the same specialized memory type as above. Again, we find the fragment $(.\text{Cons.1 as } t_c)$ at position $.\neg[0:1].*.1$, bind its location to a new symbol tail′ and recursively seek the desired subterm in it with:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((\text{tail}' \triangleleft \_ : t_{\text{list}} \text{ as } t_c), (\text{tail}_{\text{out}} : t_{\text{list}} \text{ as } t_p), \varepsilon\right)$$

This call is identical (modulo location identifiers) to the initial Seek call: we have exhibited a cycle in its call graph. In their current state, our Rebuild and Seek algorithms fail to handle such situations. △

To properly handle such cases, we must emit *recursive target code*. Naturally, we could also refuse to emit such code (in contexts when recursion is not acceptable). In both cases, we need to detect recursion.

Intuitively, a call to Seek or Rebuild leads to infinite recursion if it attempts to recursively rebuild the same combination of arguments as its own – i.e., an output value with the same type, layout and relative position from an input value with the same type, layout and pattern. This indicates that the output value contains a subterm which must be rebuilt in the exact same way as itself: the only way to emit correct code is to introduce an explicit recursive node and emit recursive calls at this position. For this purpose, we replace Seek and Rebuild with their *memoized* versions using the Wrap function defined in Algorithm 6.

```
1  function Wrap(Rebuild):
2    H := ∅
3    return H, λ({(x_i ◁ p_i : τ_i as τ̂_i) | 0 ⩽ i < n}, (x_out : τ_out as τ̂_out), u) . {
4      h ← ({(π, π', τ_i, τ̂_i, p_i) | focus (π, u) = x_i.π'}, τ_out, τ̂_out, u [x.π ↦ _])
5      if h ∈ dom (H) then
6        f ← H(h)
7        if H(h) = Uncalled(f) then H(h) := Called(f)
8        return call f(x_0, ..., x_{n-1}, x_out); success
9      else
10       f ← fresh symbol
11       H(h) := Uncalled(f)
12       ẽ ← Rebuild({(x_i ◁ p_i : τ_i as τ̂_i) | 0 ⩽ i < n}, (x_out : τ_out as τ̂_out), u)
13       if H(h) = Uncalled(f) then
14         H := H \ h
15         return ẽ
16       else if H(h) = Called(f) then
17         H(h) := Defined (f, λx_0 ... x_{n-1} x_out.ẽ)
18         return call f(x_0, ..., x_{n-1}, x_out); success
19       end
20     end
21   }
```

Algorithm 6: Wrapper for emitting recursive code (Rebuild).

```
1  function Wrap(Seek):
2    H := ∅
3    return H, λ((x_in ◁ p : τ_in as τ̂_in), (x_out : τ_out as τ̂_out), π) . {
4      h ← (τ_in, τ̂_in, p, τ_out, τ̂_out, π)
5      if h ∈ dom (H) then
6        f ← H(h)
7        if H(h) = Uncalled(f) then H(h) := Called(f)
8        return call f(x_in, x_out); success
9      else
10       f ← fresh symbol
11       H(h) := Uncalled(f)
12       ẽ ← Seek((x_in ◁ p : τ_in as τ̂_in), (x_out : τ_out as τ̂_out), π)
13       if H(h) = Uncalled(f) then
14         H := H \ h
15         return ẽ
16       else if H(h) = Called(f) then
17         H(h) := Defined (f, λx_in x_out.ẽ)
18         return call f(x_in, x_out); success
19       end
20     end
21   }
```

Algorithm 6: Wrapper for emitting recursive code (Seek).

Both Seek and Rebuild get assigned their own hashmap H, in which all calls will be recorded. Wrap memoizes each Seek or Rebuild call by hashing its *anonymized* arguments, i.e., its arguments without any input or output memory locations (Line 4). For instance, we remove variable accessors from the

valuexpression argument of Rebuild and instead keep a map from positions within the valuexpression to input value subterms. We record when we enter one of the algorithms, and generate a fresh function symbol f. If we enter this function again, we emit a call `call` $f(x_0, \ldots, x_{n-1}, x_{out})$ with the appropriate arguments and mark this function as "Called" (Line 8). Afterwards, we can use simple deforestation to get rid of extra functions. Note that, on top of emitting recursive code, this also improves sharing.

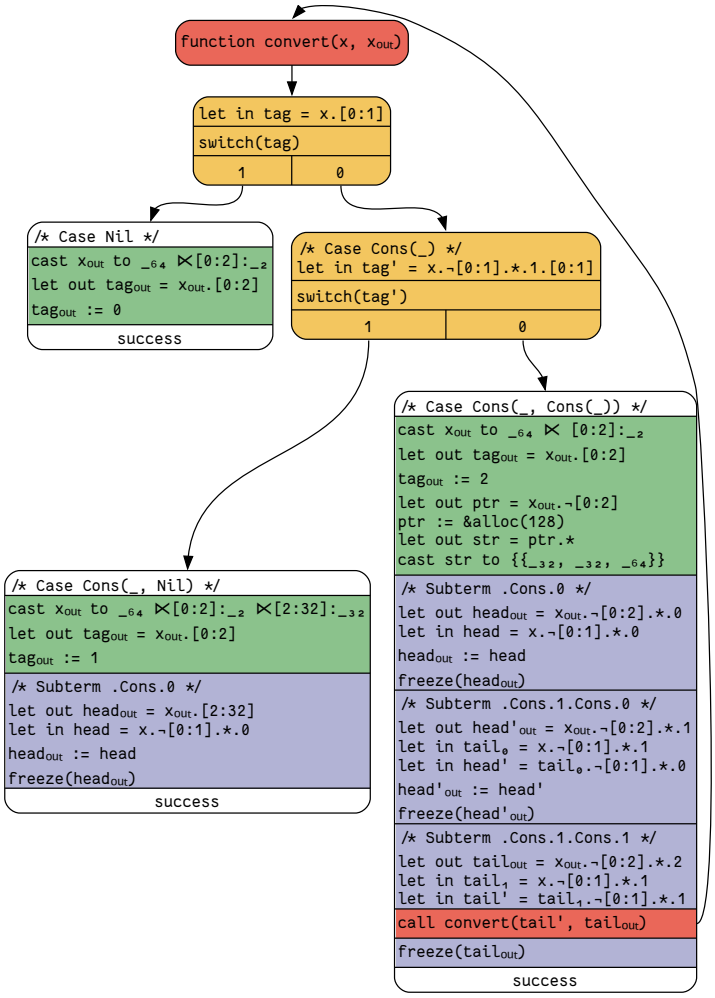


Figure 5.17: Generated code for rebuilding linked lists with a recursive `convert` function.

Example 5.23 (Recursive rebuilding of lists)**.** Recall the following cyclic Seek call from Example 5.22:

$$\textsc{Seek}_{\Delta_{\text{list}}}\left((x \triangleleft \_ : t_{\text{list}} \text{ as } t_c), (x_{\text{out}} : t_{\text{list}} \text{ as } t_p), \varepsilon\right)$$

Thanks to memoization, this call terminates and emits target code featuring a recursive function `convert` as shown in Fig. 5.17. Here, we detail the steps taken by our memoized compilation procedure to emit this recursive target code. We instantiate memoized compilation procedures and their associated hashmaps with Wrap as follows:

$$(H_{\textsc{Rebuild}}, \textsc{RecRebuild}) \triangleq \textsc{Wrap}(\textsc{Rebuild}_{\Delta_{\text{list}}}) \qquad (H_{\textsc{Seek}}, \textsc{RecSeek}) \triangleq \textsc{Wrap}(\textsc{Seek}_{\Delta_{\text{list}}})$$

We then call the memoized compilation procedure RecSeek with the same arguments:

$$\textsc{RecSeek}\left((x \triangleleft \_ : t_{\text{list}} \text{ as } t_c), (x_{\text{out}} : t_{\text{list}} \text{ as } t_p), \varepsilon\right)$$

Its anonymized argument tuple, which characterizes the task of converting a list from $\widehat{\tau}_c$ to $\widehat{\tau}_p$, is:

$$h = \left((t_{\text{list}}, t_c, \_), (t_{\text{list}}, t_p), \varepsilon\right)$$

At this point, H_{Seek} is still empty (and thus does not contain h). We add h to H_{Seek} and associate it with a new function symbol `convert`, which we initially mark as *Uncalled*. We then proceed with the actual Seek call, whose steps have been detailed in Example 5.22. The code emitted for the Nil and Cons($\langle \_, \text{Nil} \rangle$) branches is unchanged from the unwrapped version, since no Seek or Rebuild call is encountered more than once. For the third branch Cons($\langle \_, \text{Cons}(\_) \rangle$), the code emitted for shape refinement and for the first two subterms .Cons.0 and .Cons.1.Cons.0 is also unchanged.

Let us now focus on the last subterm of this branch .Cons.1.Cons.1, which is the tail of the list. We seek it within x with

$$\textsc{RecSeek}\left((x \triangleleft \text{Cons}(\langle \_, \text{Cons}(\_) \rangle) : \text{Cons}(\langle I_{32}, \text{Cons}(\dots) \rangle) \text{ as } \&_{64}(\dots) \ltimes [0:1] : (0)_1), (\text{tail}_{\text{out}} : t_{\text{list}} \text{ as } t_p), .\text{Cons.1.Cons.1}\right)$$

Following the same first steps as the unwrapped version of Seek, we find a fragment representing this subterm as t_c at position $.\neg[0:1].*.1.\neg[0:1].*.1$ within x. We bind this memory location to a new symbol tail′ and attempt to seek the desired subterm with:

$$\textsc{RecSeek}\left((\text{tail}' \triangleleft \_ : \tau_{\text{list}} \text{ as } t_c), (\text{tail}_{\text{out}} : t_{\text{list}} \text{ as } t_p), \varepsilon\right)$$

The anonymized arguments of this call are identical to those of the initial RecSeek call: they are present in the hashmap H_{Seek} and associated with the `convert` function symbol. We do not proceed with the recursive Seek call, and instead emit a function call to `convert` shown in red in the CFG to recursively rebuild the tail of the list. We mark this function as *Called* and return to the toplevel RecSeek call, which finalizes the recursive node by defining the function `convert` using the returned target expression as its body. △

5.6.2 Complete compilation algorithm

We are, at last, equipped with all necessary tools to compile the full language of ribbit expressions. We now combine all of the individual algorithms presented in this chapter into a single unified compilation procedure for the complete Ribbitulus.

In Algorithm 7, we define our general compilation procedure Compile for source expressions. Given n input descriptions characterizing already-built values, a destination x_{out} and a source expression e, it proceeds by case analysis on e to emit a target expression which evaluates e and stores its result in x_{out}. Compilation of pivot expressions is delegated to our dedicated procedures Seek and Rebuild defined in Section 5.5, with the former being used for accessors and the latter for other forms of valuexpressions. Both of these functions are hashconsed with Wrap to handle situations which require recursive target code: we use their wrapped versions RecSeek and RecRebuild, which are instantiated in our toplevel compilation procedure CompileProg. Similarly, compilation of pattern matching is delegated to the Destruct procedure described in Section 5.3 after each right-hand side expression has been recursively Compiled. The compilation of remaining source language constructs is straightforward. For let-bindings, we emit code which allocates an adequately-sized new memory location, fills its contents by evaluating the let-bound expression, then freezes it for later use before evaluating the rest of the expression. Finally, source function calls are immediately translated to target function calls with the same function symbol and input arguments, using x_{out} as the destination argument.

Data: Δ the type variable environment
Data: RecSeek the memoized version of $\textsc{Seek}_\Delta$
Data: RecRebuild the memoized version of $\textsc{Rebuild}_\Delta$
Data: n input descriptions $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)$ for free variables in e
Data: x_{out} the destination location
Data: e the source expression
Result: Target expression $\tilde{e}$ storing its result in x_{out}
function $\textsc{Compile}_\Delta(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, x_{out}, e)$**:**
  cases e **:**
    case $(x_i.\pi : \tau \text{ as } \widehat{\tau})$ **:**
      return RecSeek $((x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i), (x_{out} : \tau \text{ as } \widehat{\tau}), \pi)$
    case $(u : \tau \text{ as } \widehat{\tau})$ **:**
      return RecRebuild $(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau \text{ as } \widehat{\tau}), u)$
    case *let* $x : (\tau \text{ as } \widehat{\tau}) = e_0$ *in* e_1 **:**
      $\tilde{e}_0 \leftarrow \textsc{Compile}_\Delta(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, x, e_0)$
      $\tilde{e}_1 \leftarrow \textsc{Compile}_\Delta(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\} \cup \{(x \triangleleft \_ : \tau \text{ as } \widehat{\tau})\}, x_{out}, e_1)$
      return `let out` $x =$ `alloc`$(|\widehat{\tau}|)$; $\tilde{e}_0$; `freeze`(x); $\tilde{e}_1$
    case $f(x_i)$ **:**
      return `call` $f(x_i, x_{out})$; `success`
    case *match*$(x_i)\{p_j \rightarrow e_j \mid 0 \leqslant j < N\}$ **:**
      for $i' \in \{0, \ldots, n-1\} \setminus \{i\}$ **do** $p'_{i'} \leftarrow p_{i'}$
      for $j \in \{0, \ldots, N-1\}$ **do**
        $p'_i \leftarrow p_j$
        $\tilde{e}_j \leftarrow \textsc{Compile}_\Delta(\{(x_{i'} \triangleleft p'_{i'} : \tau_{i'} \text{ as } \widehat{\tau}_{i'}) \mid 0 \leqslant i' < n\}, x_{out}, e_j)$
      return $\textsc{Destruct}_\Delta((x_i : \widehat{\tau}_i), \{(p_j, \tilde{e}_j) \mid 0 \leqslant j < N\})$

Algorithm 7: Main compilation procedure for expressions: Compile. RecSeek and RecRebuild refer to the wrapped versions of Seek and Rebuild respectively, which we define in Algorithm 8.

Our outermost compilation interface CompileProg is presented in Algorithm 8. Its input is a complete ribbit program represented by a function environment Σ, a typing environment Γ, an identifier x and a toplevel source expression e of type τ and layout $\widehat{\tau}$. This procedure is responsible for initializing the hashmaps $H_{\textsc{Seek}}$ and $H_{\textsc{Rebuild}}$, compiling each source function's body as well as the toplevel expression with Compile, and adding the recursive functions defined in Wrap to the target function environment.

Data: Δ the type variable environment
Data: Γ the (function) typing environment
Data: Σ the source function environment
Data: $(x : \tau \text{ as } \widehat{\tau})$ the output description for the toplevel result
Data: e the toplevel source expression
Result: $\tilde{\Sigma}$ the target function environment (includes compiled versions of Σ functions)
Result: $\tilde{e}$ the toplevel target expression (compiled version of e)
function $\textsc{CompileProg}_\Delta(\Gamma, \Sigma, (x : \tau \text{ as } \widehat{\tau}), e)$**:**
  $(H_{\textsc{Seek}}, \textsc{RecSeek}) := \textsc{Wrap}(\textsc{Seek}_\Delta)$
  $(H_{\textsc{Rebuild}}, \textsc{RecRebuild}) := \textsc{Wrap}(\textsc{Rebuild}_\Delta)$
  $\tilde{\Sigma} := \emptyset$
  for $(f \mapsto \lambda x_f.e_f) \in \tilde{\Sigma}$ **do**
    $((\tau_{in} \text{ as } \widehat{\tau}_{in}) \to (\tau_{out} \text{ as } \widehat{\tau}_{out})) \leftarrow \Gamma(f)$
    $\tilde{e}_f \leftarrow \textsc{Compile}_\Delta(\{(x_f \triangleleft \_ : \tau_{in} \text{ as } \widehat{\tau}_{in})\}, x, e_f)$
    $\tilde{\Sigma} := \tilde{\Sigma} \cup \{(f \mapsto \lambda x_f x.\tilde{e}_f)\}$
  $\tilde{e} \leftarrow$ `let out` $x =$ `alloc`$(|\widehat{\tau}|)$; $\textsc{Compile}_\Delta(\emptyset, x, e)$; `freeze`$(x)$; `success`
  for $(h \mapsto \mathit{Defined}(f, \lambda x_0 \ldots x_{n-1} x_{out}.\tilde{e})) \in H_{\textsc{Seek}} \cup H_{\textsc{Rebuild}}$ **do**
    $\tilde{\Sigma} := \tilde{\Sigma} \cup \{(f \mapsto \lambda x_0 \ldots x_{n-1} x_{out}.\tilde{e})\}$
  return $(\tilde{\Sigma}, \tilde{e})$

Algorithm 8: $\textsc{CompileProg}$ – main compilation interface handling both functions and toplevel expressions.

We finally have a single compilation procedure for the Ribbitulus. The following example illustrates the compilation of a program with multiple functions manipulating RISC-V instructions.

Example 5.24 (Compilation of a program with RISC-V instructions)**.** Recall the following collection of types, functions and expressions from Example 5.4:

$$\tau_{bool} = \text{True} \mid \text{False} \qquad \widehat{\tau}_{bool} = \text{split}(\varepsilon) \left\{ \begin{array}{lll} 0 & \text{from} & \text{False} \Rightarrow (0)_8 \\ 1 & \text{from} & \text{True} \Rightarrow (1)_8 \end{array} \right\}$$

$$\Gamma = \left\{ \begin{array}{lll} \texttt{is\_compressible} & : & (\tau_{riscv} \text{ as } \widehat{\tau}_{riscv} \to \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \texttt{is\_nonzero\_register} & : & (\tau_{reg} \text{ as } \widehat{\tau}_{reg} \to \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \texttt{is\_popular\_register} & : & (\tau_{reg} \text{ as } \widehat{\tau}_{reg} \to \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \end{array} \right\}$$

$$\Sigma = \left\{ \begin{array}{lll} \texttt{is\_compressible} & \mapsto \lambda x. & e_c \\ \texttt{is\_nonzero\_register} & \mapsto \lambda x. & e_{0reg} \\ \texttt{is\_popular\_register} & \mapsto \lambda x. & e_{preg} \end{array} \right\} \qquad e_{0reg} = \text{match}(x) \left\{ \begin{array}{lll} X_0 & \to & (\text{False} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \_ & \to & (\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \end{array} \right\}$$

$$e_{preg} = \text{match}(x) \left\{ \begin{array}{lll} X_8 & \to & (\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \ldots & \to & \ldots \\ X_{15} & \to & (\text{True} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \\ \_ & \to & (\text{False} : \tau_{bool} \text{ as } \widehat{\tau}_{bool}) \end{array} \right\}$$

e_c = match(x) {

$$
\begin{array}{ll}
\mathrm{Jal}(\langle X_1, \_\rangle) & \to \mathsf{let}\ n : I_{20}\ \mathsf{as}\ I_{20} = x.\mathrm{Jal}.1\ \mathsf{in} \\
& \quad (n < 4096 : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}}) \\
\mathrm{Add}(\langle \_, \_, \_\rangle) & \to \mathsf{let}\ r_d : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Add}.0\ \mathsf{in} \\
& \quad \mathsf{let}\ r_{s1} : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Add}.1\ \mathsf{in}\ \mathsf{let}\ b_{s1} : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}} = \texttt{is\_nonzero\_register}(r_{s1})\ \mathsf{in} \\
& \quad \mathsf{let}\ r_{s2} : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Add}.2\ \mathsf{in}\ \mathsf{let}\ b_{s2} : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}} = \texttt{is\_nonzero\_register}(r_{s2})\ \mathsf{in} \\
& \quad (r_d = r_{s1} \wedge b_{s1} \wedge b_{s2} : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}}) \\
\mathrm{Addi}(\langle \_, \_, \_\rangle) & \to \mathsf{let}\ r_d : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Addi}.0\ \mathsf{in} \\
& \quad \mathsf{let}\ r_s : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Addi}.1\ \mathsf{in}\ \mathsf{let}\ b : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}} = \texttt{is\_nonzero\_register}(r_s)\ \mathsf{in} \\
& \quad \mathsf{let}\ n : I_{12}\ \mathsf{as}\ I_{12} = x.\mathrm{Addi}.2\ \mathsf{in} \\
& \quad (r_d = r_s \wedge b \wedge n < 64 : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}}) \\
\mathrm{Sw}(\langle \_, \_, \_\rangle) & \to \mathsf{let}\ r_0 : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Sw}.0\ \mathsf{in}\ \mathsf{let}\ b_0 : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}} = \texttt{is\_popular\_register}(r_0)\ \mathsf{in} \\
& \quad \mathsf{let}\ r_1 : \tau_{\mathrm{reg}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{reg}} = x.\mathrm{Sw}.1\ \mathsf{in}\ \mathsf{let}\ b_1 : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}} = \texttt{is\_popular\_register}(r_1)\ \mathsf{in} \\
& \quad \mathsf{let}\ n : I_{12}\ \mathsf{as}\ I_{12} = x.\mathrm{Sw}.2\ \mathsf{in} \\
& \quad \mathsf{let}\ n_l : I_5\ \mathsf{as}\ I_5 = n.[0:5]\ \mathsf{in}\ \mathsf{let}\ n_h : I_2\ \mathsf{as}\ I_2 = n.[10:2]\ \mathsf{in} \\
& \quad (b_0 \wedge b_1 \wedge n_l = 0 \wedge n_h = 0 : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}}) \\
\_ & \to (\mathrm{False} : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}})
\end{array}
$$

}

Consider the following toplevel expression, which builds a value representing a store-word instruction as in Example 5.5, then calls the `is_compressible` function to determine whether it is compressible:

$$e = \mathsf{let}\ \texttt{instr} : \tau_{\mathrm{riscv}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{riscv}} = \mathrm{Sw}(\langle X_1, X_2, 42\rangle)\ \mathsf{in}\ \texttt{is\_compressible(instr)}$$

We compile this complete program with

$$\textsc{CompileProg}\,(\Gamma, \Sigma, (\texttt{res} : \tau_{\mathrm{bool}}\ \mathsf{as}\ \widehat{\tau}_{\mathrm{bool}}), e)$$

which yields the CFG forest depicted in Fig. 5.18. In this forest, each compiled function corresponds to a pink node leading to its body. Note that the specification of a compilation scheme for primitive operations – in this example, on booleans and integers – is implementation-dependent; here, we assume that comparison and "AND" operators of arbitrary arity are available. The compilation rules defined by Compile are clearly visible in the different parts of each CFG. △

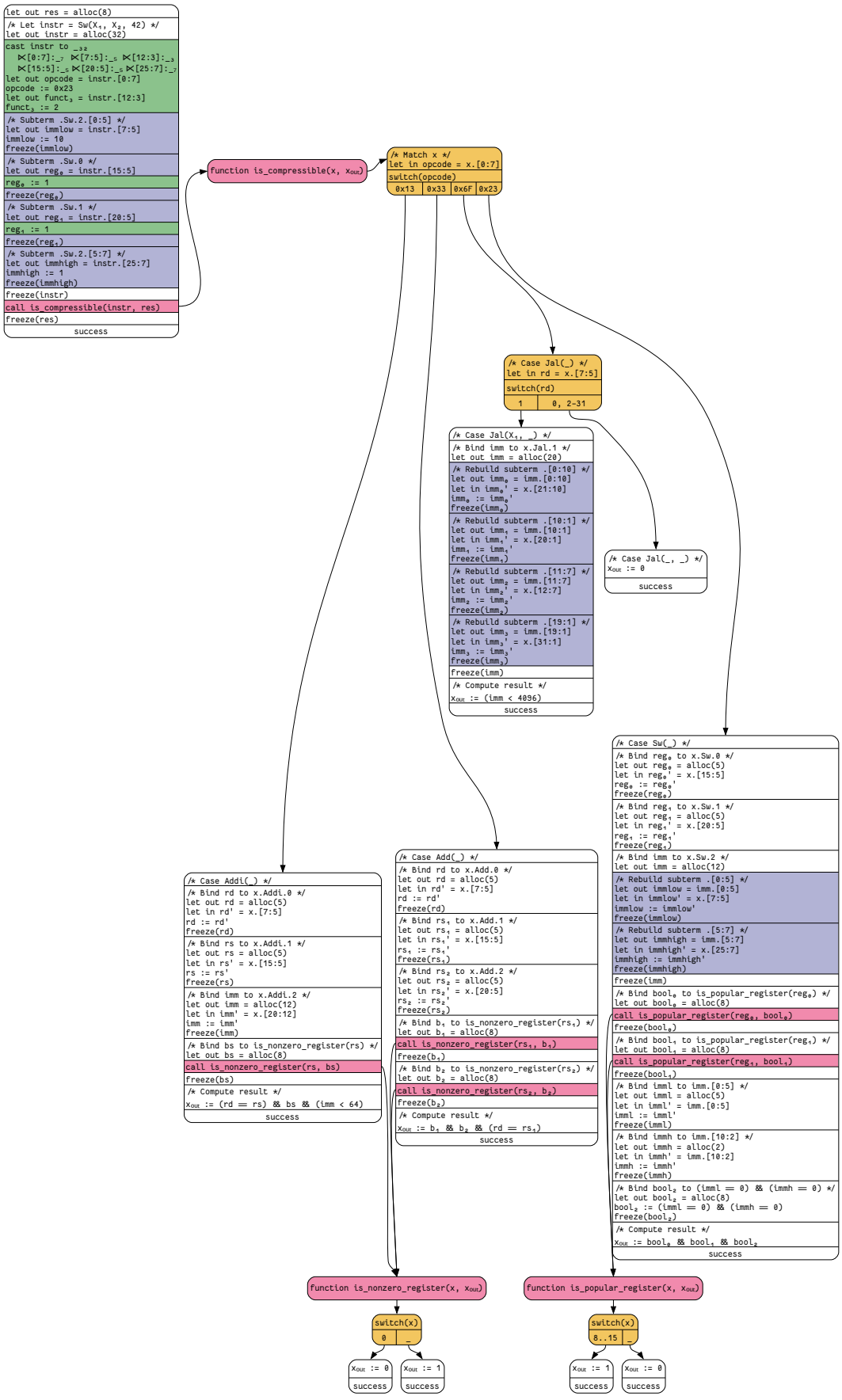


Figure 5.18: Output of CompileProg for the program of Example 5.24.

5.7 Metatheory

We now prove our compilation approach correct by exhibiting a *weak simulation* relation between the source memory evaluation $\hookrightarrow$ from Section 3.3.2 and the target evaluation $\rightsquigarrow$ from Section 5.2.2. Combined with Theorem 3.3, this gives us a weak simulation between source and target evaluation, showing correctness of the Compile procedure.

The key idea is to modify our compilation algorithm to emit synchronization tokens in the target program to synchronize its execution with source evaluation. All other steps from the target evaluation $\rightsquigarrow$ are $\not\vdash$-transitions. Tokens may either appear by themselves, in which case the corresponding transition has no effect on the underlying target evaluation state, or be attached to another target instruction – for instance $x := c$[EConstant-token]; ... – in which case the normal semantics of this instruction applies.

Note that our target semantics $\rightsquigarrow$ is deterministic, whereas the memory-level source semantics $\hookrightarrow$ is non-deterministic. To prove that the latter simulates the former, we show that $\hookrightarrow$ can always take a step which corresponds to the program order determined by our compilation algorithms.

5.7.1 Relation on constants and Refine

We start with the $\tilde{\mathcal{R}}_{\text{const}}$ relation defined in Fig. 5.20, which characterizes target code emitted by Refine. This relation also allows us to demonstrate our approach in a somewhat restricted context. An annotated version of Refine is shown in Fig. 5.19. The new proof-exclusive elements are shown in purple. We simply emit specific tokens for allocation and cast operations.

We then define a simulation relation $\tilde{\mathcal{R}}_{\text{consts}}$ in Fig. 5.20 for target code building the "constant" parts of memory values, i.e., target code which only allocates, casts and writes constants in memory. We will use this simulation to prove the correctness of Refine.

Given an output location x_{out}, a memory environment ς_{src}, a target location environment $\tilde{\sigma}_{\text{out}}$ and target memory environment ς_{tgt}, we have: $x_{\text{out}}, \widehat{p}_{\text{new}} \vdash (\varsigma_{\text{src}}, \widehat{u})\ \tilde{\mathcal{R}}_{\text{consts}}(\tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}}, \tilde{e})$ if x_{out} points to a value of shape $\widehat{p}_{\text{old}} = \textbf{shape\_of}_{\varsigma_{\text{src}}}(\widehat{u})$ in ς_{tgt} before execution of $\tilde{e}$, and of shape $\widehat{p}_{\text{new}}$ after.

Its base rule IRRConstsRefine corresponds exactly to the application of Refine. The other rules corresponds to cases where the toplevel constructor of $\widehat{p}_{\text{old}}$ and $\widehat{p}_{\text{new}}$ are identical, which proceed by induction.

$\textsc{Refine}_{\Delta}(x)\{$

$\widehat{p}$	, $\widehat{p}$	$\longrightarrow$	`success`
$\__{\ell}$	, $(c)_{\ell}$	$\longrightarrow$	$x := c$[EConstant-token]; `success`
$\__{\ell}$	, $\&_{\ell}(\widehat{p})$	$\longrightarrow$	$x :=$ `&alloc`$(\lvert\widehat{p}\rvert)$[EPointer-token]; $\textsc{Refine}\left(x, \&_{\ell}\left(\__{\lvert\widehat{p}\rvert}\right), \&_{\ell}(\widehat{p})\right)$
$\__{\ell}$	, $\widehat{p} \underset{0\leqslant i<n}{\bowtie} [o_i : \ell_i] : \widehat{p}_i$	$\longrightarrow$	`cast` x `to` $\__{\ell} \underset{0\leqslant i<n}{\bowtie} [o_i : \ell_i] : \__{\ell_i}$[EComposite-token]; $\textsc{Refine}\left(x, \__{\ell} \underset{0\leqslant i<n}{\bowtie} [o_i : \ell_i] : \__{\ell_i}, \widehat{p} \underset{0\leqslant i<n}{\bowtie} [o_i : \ell_i] : \widehat{p}_i\right)$
$\__{\ell}$	, $\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	$\longrightarrow$	`cast` x `to` $\left\{\!\!\left\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\right\}\!\!\right\}$[EStruct-token]; $\textsc{Refine}\left(x, \left\{\!\!\left\{\__{\lvert\widehat{p}_0\rvert}, \ldots, \__{\lvert\widehat{p}_{n-1}\rvert}\right\}\!\!\right\}, \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}\right)$
$\&_{\ell}(\widehat{p})$	, $\&_{\ell}(\widehat{p}')$	$\longrightarrow$	`let out` $x' = x.*$; $\textsc{Refine}(x', \widehat{p}, \widehat{p}')$
$\widehat{p} \underset{0\leqslant i<n}{\bowtie} r_i : \widehat{p}_i$	, $\widehat{p}' \underset{0\leqslant i<n}{\bowtie} r_i : \widehat{p}'_i$	$\longrightarrow$	`let out` $x' = x.\neg r_0 \ldots \neg r_{n-1}$; `let out` $x_0 = x.r_0$; ...; `let out` $x_{n-1} = x.(n-1)$; $\textsc{Refine}(x', \widehat{p}, \widehat{p}')$; $\textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0)$; ...; $\textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$
$\{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\}$	, $\{\!\{\widehat{p}'_0, \ldots, \widehat{p}'_{n-1}\}\!\}$	$\longrightarrow$	`let out` $x_0 = x.0$; ...; `let out` $x_{n-1} = x.(n-1)$; $\textsc{Refine}(x_0, \widehat{p}_0, \widehat{p}'_0)$; ...; $\textsc{Refine}(x_{n-1}, \widehat{p}_{n-1}, \widehat{p}'_{n-1})$
$\widehat{p}$	, $\widehat{p}'$	$\longrightarrow$	`fail`

$\}$

Figure 5.19: Version of Refine augmented with tokens.

IRRConstsRefine

$$\frac{(x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad \widehat{p}_{old} = \mathbf{shape\_of}_{\varsigma_{src}}(\widehat{u}) \qquad \mathbf{shape\_of}_{\varsigma_{tgt}}\left(\widehat{\mathbf{focus}}_{\varsigma_{tgt}}\left(\widehat{\pi}_{out}, \varsigma_{tgt}(a_{out})\right)\right) = \widehat{p}_{old} \qquad \textsc{Refine}\left(x_{out}, \widehat{p}_{old}, \widehat{p}_{new}\right) \neq \texttt{fail}}{x_{out}, \widehat{p}_{new} \vdash (\varsigma_{src}, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}\ \left(\tilde{\sigma}_{out}, \varsigma_{tgt}, \textsc{Refine}\left(x_{out}, \widehat{p}_{old}, \widehat{p}_{new}\right)\right)}$$

IRRConstsPointer

$$\frac{(x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad (x'_{out} \mapsto a_{out}.\widehat{\pi}_{out}.*) \in \tilde{\sigma}_{out} \qquad x'_{out}, \widehat{p}_{new} \vdash (\varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \tilde{e})}{x_{out}, \&_\ell\ (\widehat{p}_{new}) \vdash (\varsigma, \&_\ell\ (\widehat{u}))\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \tilde{e})}$$

IRRConstsComposite

$$\frac{(x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad (x'_{out} \mapsto a_{out}.\widehat{\pi}_{out}.\neg r_0 \ldots \neg r_{n-1}) \in \tilde{\sigma}_{out} \qquad x'_{out}, \widehat{p} \vdash (\varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}\ (\tilde{\sigma}_{out}, \tilde{e}) \qquad (x_{out,i} \mapsto a_{out}.\widehat{\pi}_{out}.r_i) \in \tilde{\sigma}_{out} \qquad x_{out,i}, \widehat{p}_i \vdash (\varsigma, \widehat{u}_i)\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \tilde{e}_i)}{x_{out}, \widehat{p} \bowtie_{0 \leqslant i < n} r_i : \widehat{p}_i \vdash \left(\varsigma, \widehat{u} \bowtie_{0 \leqslant i < n} r_i : \widehat{u}_i\right)\ \tilde{\mathcal{R}}_{\ consts}\ (\tilde{\sigma}_{out}, \tilde{e}; \tilde{e}_0; \ldots; \tilde{e}_{n-1})}$$

IRRConstsStruct

$$\frac{(x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad (x_{out,i} \mapsto a_{out}.\widehat{\pi}_{out}.i) \in \tilde{\sigma}_{out} \qquad x_{out,i}, \widehat{p}_i \vdash (\varsigma, \widehat{u}_i)\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \tilde{e}_i)}{x_{out}, \{\!\{\widehat{p}_0, \ldots, \widehat{p}_{n-1}\}\!\} \vdash (\varsigma, \{\!\{\widehat{u}_0, \ldots, \widehat{u}_{n-1}\}\!\})\ \tilde{\mathcal{R}}_{\ consts}\ (\tilde{\sigma}_{out}, \tilde{e}_0; \ldots; \tilde{e}_{n-1})}$$

Figure 5.20: Relation characterizing Refine intermediate states: $x_{out}, \widehat{p}_{new} \vdash (\varsigma_{src}, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e})$.

Lemma 5.1 (Refine). *Let* $\Delta, \Gamma, \widehat{\tau}, \widehat{p}_{new}, \varsigma_{src}, \widehat{u}, x_{out}, a_{out}, \widehat{\pi}_{out}, \tilde{\sigma}_{out}, \varsigma_{tgt}$ *and* $\tilde{e}$ *such that*

$$\vDash \Delta, \Gamma, \varsigma_{src} \vdash \widehat{u} : \widehat{\tau} \qquad (x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad \widehat{\mathbf{focus}}_{\varsigma_{tgt}}\left(\widehat{\pi}_{out}, \varsigma_{tgt}(a_{out})\right) = \widehat{v}_{out}$$

$$x_{out}, \widehat{p}_{new} \vdash (\varsigma_{src}, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e})$$

We either have

$$\mathbf{shape\_of}_{\varsigma_{src}}(\widehat{u}) = \mathbf{shape\_of}_{\varsigma_{tgt}}\left(\widehat{\mathbf{focus}}_{\varsigma_{tgt}}\left(\widehat{\pi}_{out}, \varsigma_{tgt}(a_{out})\right)\right) = \widehat{p}_{new} \qquad \tilde{e} = \texttt{success}$$

or there exist $\tilde{\sigma}'_{out}$ *and* $\tilde{e}'$ *such that for any* $\tilde{\Sigma}$, ρ, $\tilde{\sigma}_{in}$,

$$\tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e} \rightsquigarrow_{\succeq} \rho, \tilde{\sigma}_{in}, \tilde{\sigma}'_{out}, \varsigma_{tgt}, \tilde{e}' \qquad (x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}'_{out}$$

$$x_{out}, \widehat{p}_{new} \vdash (\varsigma_{src}, \widehat{u})\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}'_{out}, \varsigma_{tgt}, \tilde{e}')$$

or there exist a label L, ς'_{src}, $\widehat{u}'$, $\tilde{\sigma}'_{out}$, ς'_{tgt} *and* $\tilde{e}'$ *such that for any* Σ, $\widehat{\sigma}$, $\tilde{\Sigma}$, ρ, $\tilde{\sigma}_{in}$,

$$\Delta, \Sigma \vdash \Gamma, \widehat{\sigma}, \varsigma_{src}, \widehat{u} \mapsto_L \Gamma, \widehat{\sigma}, \varsigma'_{src}, \widehat{u}' \qquad \tilde{\Sigma} \vdash \rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e} \rightsquigarrow_L \rho, \tilde{\sigma}_{in}, \tilde{\sigma}'_{out}, \varsigma'_{tgt}, \tilde{e}'$$

$$(x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}'_{out} \qquad x_{out}, \widehat{p}_{new} \vdash (\varsigma'_{src}, \widehat{u}')\ \tilde{\mathcal{R}}_{\ consts}(\tilde{\sigma}'_{out}, \varsigma'_{tgt}, \tilde{e}')$$

Proof. By induction on $\tilde{\mathcal{R}}_{\ consts}$.

IRRConstsRefine: let

$$\widehat{p}_{old} = \mathbf{shape\_of}_{\varsigma_{src}}(\widehat{u})$$

We have

$$\mathbf{shape\_of}_{\varsigma_{tgt}}(\widehat{v}_{out}) = \widehat{p}_{old} \qquad \tilde{e} = \textsc{Refine}(x_{out}, \widehat{p}_{old}, \widehat{p}_{new}) \neq \texttt{fail}$$

We proceed by case analysis on $\tilde{e}$:

Refine **success case** If $\widehat{p}_{\text{old}} = \widehat{p}_{\text{new}}$, we have $\tilde{e} = \texttt{success}$.

Refine **token cases** We have $\widehat{p}_{\text{old}} = \__\ell$. For simplicity, we only consider the following case:

$$\widehat{u} = (u : \tau \text{ as } (c)_\ell) \qquad \widehat{p}_{\text{new}} = (c)_\ell \qquad \tilde{e} = x_{\text{out}} := c[\textsc{EConstant}\text{-token}]; \texttt{success}$$

The source and target programs $\widehat{u}$ and $\tilde{e}$ both go through a EConstant step (using the IREWriteConstant rule for $\tilde{e}$), yielding normal forms related to each other by $\tilde{\mathcal{R}}_{\text{consts}}$ (IRRConstsRefine rule):

$$\Gamma, \widehat{\sigma}, \varsigma_{\text{src}}, (u : \tau \text{ as } (c)_\ell) \hookrightarrow \Gamma, \widehat{\sigma}, \varsigma_{\text{src}}, (c)_\ell \qquad \tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}}, \tilde{e} \rightsquigarrow \tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}}[a_{\text{out}}.\widehat{\pi}_{\text{out}} \leftarrow (c)_\ell], \texttt{success}$$

Refine **non-token cases** For simplicity, we only consider the following case:

$$\widehat{p}_{\text{old}} = \&_\ell\,(\widehat{p}'_{\text{old}}) \qquad \widehat{u} = \&_\ell\,(\widehat{u}') \qquad \widehat{p}_{\text{new}} = \&_\ell\,(\widehat{p}'_{\text{new}})$$

$$\tilde{e} = \texttt{let out } x'_{\text{out}} = x_{\text{out}}.*; \textsc{Refine}(x'_{\text{out}}, \widehat{p}'_{\text{old}}, \widehat{p}'_{\text{new}})$$

Induction hypothesis: the result holds for $\widehat{u}'$, $\widehat{p}'_{\text{new}}$ and $\textsc{Refine}(x'_{\text{out}}, \widehat{p}'_{\text{old}}, \widehat{p}'_{\text{new}})$. The target program $\tilde{e}$ goes through a $\trianglelefteq$-transition (IRESubOutLoc) binding the new symbol x'_{out} to the pointee memory value, and we conclude using the induction hypothesis and the IRRConsts-Pointer rule:

$$\tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}}, \tilde{e} \rightsquigarrow_{\trianglelefteq} \tilde{\sigma}_{\text{out}} \cup \{x'_{\text{out}} \mapsto a_{\text{out}}.\widehat{\pi}_{\text{out}}.*\}, \varsigma_{\text{tgt}}, \textsc{Refine}(x'_{\text{out}}, \widehat{p}'_{\text{old}}, \widehat{p}'_{\text{new}})$$

IRRConstsPointer, IRRConstsComposite, IRRConstsStruct: immediate from induction hypotheses.

□

5.7.2 Annotated compilation code

function $\textsc{Rebuild}_\Delta\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), u\right)$**:**
  // Explicitly unroll type variables.
  if $\widehat{\tau}_{out} = t \in \mathit{TyVars}$ **then**
    $\tilde{e} \leftarrow \textsc{Rebuild}_\Delta\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau_{out} \text{ as } \Delta(t)), u\right)$
    return *ETypeVar-token*; $\tilde{e}$
  // Base case: target value is a constant encoded as a primitive type.
  else if $u = c \wedge \widehat{\tau}_{out} = I_\ell$ **then return** $x_{out} := c$[*EAtom-token*]; `success`
  else // Otherwise, explore all cases of the output value.
    // Insert ESplit-tokens before accessing branch-specific locations.
    $\tilde{e}_{consts} \leftarrow \textsc{Refine}\left(x_{out}, \_{\mid\widehat{\tau}_{out}\mid}, \textbf{shape\_of}_\Delta(\widehat{\tau}_{out})\right)$
    $\widehat{\Pi}_{splits} \leftarrow \left\{\widehat{\pi} \;\middle|\; \widehat{\textbf{focus}}_\Delta\left(\widehat{\pi}, \widehat{\tau}_{out}\right) = \text{split}\left(\ldots\right)\right\}$
    $\tilde{e}_{splits} \leftarrow$ **for** $\widehat{\pi} \in \widehat{\Pi}_{splits}$ **do yield** *ESplit-token*
    $\text{Pos}_{out} \leftarrow \{(x.\pi, \pi') \mid \textbf{focus}\left(\pi', u\right) = x.\pi\}$
    $p_{out} \leftarrow \textbf{Remap}\left(\{x_i \mapsto p_i \mid 0 \leqslant i < n\}, u\left[x.\pi \mapsto \_\right], \text{Pos}_{out}\right)$
    $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, \mathit{frags}_b \in \textbf{Explore}_\Delta(p_{out}, \tau_{out}, \widehat{\tau}_{out})$ **do**
      // Allocate memory, cast and fill in constant parts as needed for this memory type.
      $\tilde{e}'_{consts} \leftarrow \textsc{Refine}\left(x_{out}, \textbf{shape\_of}_\Delta(\widehat{\tau}_{out}), \textbf{shape\_of}_\Delta(\widehat{\tau}_b)\right)$
      // Allow EWord steps for definitively uninitialized words.
      $\widehat{\Pi}_{words} \leftarrow \left\{\widehat{\pi} \;\middle|\; \widehat{\textbf{focus}}_\Delta\left(\widehat{\pi}, \widehat{\tau}_b\right) = \__\ell\right\}$
      $\tilde{e}_{words} \leftarrow$ **for** $\widehat{\pi} \in \widehat{\Pi}_{words}$ **do yield** *EWord-token*
      // Allow EAddress steps for new pointers.
      $\widehat{\Pi}_{addrs} \leftarrow \left\{\widehat{\pi} \;\middle|\; \widehat{\textbf{focus}}\left(\widehat{\pi}, \textbf{shape\_of}_\Delta(\widehat{\tau}_b)\right) = \&_\ell\left(\ldots\right)\right\}$
      $\tilde{e}_{addrs} \leftarrow$ **for** $\widehat{\pi} \in \widehat{\Pi}_{addrs}$ **do yield** *EAddress-token*
      // Rebuild target fragments from input values, which we specialize for the current branch.
      for $i \in \{0, \ldots, n-1\}$ **do**
        $\text{Pos}_i \leftarrow \{(x_{out}.\pi, \pi') \mid \textbf{focus}\left(\pi, u\right) = x_i.\pi'\}$
        $p'_i \leftarrow \textbf{Remap}(\{x_{out} \mapsto p_b\}, p_i, \text{Pos}_i)$
      $\tilde{e}_{frags} \leftarrow$ **for** $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathit{frags}_b$ **do**
        $x_f \leftarrow$ fresh symbol
        $\tau_f \leftarrow \textbf{focus}\left(\pi_f, \tau_b\right)$
        if $\exists 0 \leqslant i < n, \exists(x_i.\pi_{in}, \pi_{out}) \in \mathit{Pos}_{out}, \exists \pi, \pi_{out}.\pi = \pi_f$ **then**
          // If this fragment corresponds to a single piece of an input value, Seek it within this value.
          $\tilde{e}_f \leftarrow \textsc{Seek}((x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i), (x_f : \tau_f \text{ as } \widehat{\tau}_f), \pi_{in}.\pi)$
        else // Otherwise, Rebuild it from smaller pieces.
          $\tilde{e}_f \leftarrow \textsc{Rebuild}(\{(x_i \triangleleft p'_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_f : \tau_f \text{ as } \widehat{\tau}_f), \textbf{focus}\left(\pi_f, u\right))$
        $\tilde{e}_t \leftarrow$ **if** $\widehat{\textbf{focus}}_\Delta\left(\widehat{\pi}_f, \widehat{\tau}_b\right) = I_\ell$ **then** `success`
        else EFragment-token; `success`
        yield `let out` $x_f = x_{out}.\widehat{\pi}_f$; $\tilde{e}_t$; $\tilde{e}_f$
      yield $(\{x_i \mapsto p'_i \mid 0 \leqslant i < n\}, \tilde{e}_{consts}; \tilde{e}_{splits}; \tilde{e}'_{consts}; \tilde{e}_{words}; \tilde{e}_{frags}; \tilde{e}_{addrs})$
    // Assemble these branches into a decision tree.
    return $\textsc{Destruct}_\Delta\left(\{x_i : \widehat{\tau}_i \mid 0 \leqslant i < n\}, B\right)$

Algorithm 9: Version of Rebuild adapted for a less painful proof.

function $\textsc{Seek}_\Delta((x_{in} \triangleleft p_{in} : \tau_{in} \text{ as } \widehat{\tau}_{in}), (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), \pi)$**:**
// Invariant: π and p_{in} are compatible.
// Explicitly unroll type variables in $\widehat{\tau}_{out}$.
if $\widehat{\tau}_{out} = t \in \mathit{TyVars}$ **then**
  $\tilde{e} \leftarrow \textsc{Seek}_\Delta((x_{in} \triangleleft p_{in} : \tau_{in} \text{ as } \widehat{\tau}_{in}), (x_{out} : \tau_{out} \text{ as } \Delta(t)), \pi)$
  return *ETypeVar-token*; $\tilde{e}$
// Base case: input and output values are the same data with the same representation.
if $\pi = \varepsilon \wedge \widehat{\tau}_{in} = \widehat{\tau}_{out}$ **then return** $x_{out} := x_{in}$[*EVarAccess-token*]; `success`
else // Otherwise, Explore all cases of the input value.
  $B \leftarrow$ **for** $p_b, \tau_b, \widehat{\tau}_b, \mathit{frags}_b \in \textbf{Explore}_\Delta(p_{in}, \tau_{in}, \widehat{\tau}_{in})$ **do**
    // Seek a fragment containing the piece of data at π.
    if $\exists(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \mathit{frags}_b, \pi_f \preceq \pi$ **then**
      // Found one. We focus on it and Seek inside.
      $x_f \leftarrow$ fresh symbol
      $\tau_f, p_f \leftarrow$ **focus** (π_f, τ_b), **focus** (π_f, p_b)
      $\tilde{e} \leftarrow \textsc{Seek}((x_f \triangleleft p_f : \tau_f \text{ as } \widehat{\tau}_f), (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}),$ **focus** $(\pi_f, \pi))$
      $\tilde{e}_b \leftarrow$ `let in` $x_f = x_{in}.\widehat{\pi}_f$[EVarFocus-token]; $\tilde{e}$
    else // Otherwise, Rebuild from smaller pieces.
      if $\widehat{\tau}_{out} = I_\ell$ **then** // If we are seeking a primitive, decompose it in individual bits
        $\widehat{\tau}'_{out} \leftarrow \__\ell \bowtie_{0 \leqslant i < \ell} [i : 1] : (.[i : 1] \text{ as } I_1)$
        $\tilde{e} \leftarrow \textsc{Rebuild}(\{(x_{in} \triangleleft p_b : \tau_b \text{ as } \widehat{\tau}_b)\}, (x_{out} : \tau_{out} \text{ as } \widehat{\tau}'_{out}), x_{in}.\pi)$
        $\tilde{p} \leftarrow \__\ell \bowtie_{0 \leqslant i < \ell} [i : 1] : \__1$
        $\tilde{e}_b \leftarrow$ `cast` x_{out} `to` $\tilde{p}$[EFission-token]; $\tilde{e}$; `cast` x_{out} `to` I_ℓ; `success`
      else $\tilde{e}_b \leftarrow \textsc{Rebuild}(\{(x_{in} \triangleleft p_b : \tau_b \text{ as } \widehat{\tau}_b)\}, (x_{out} : \tau_{out} \text{ as } \widehat{\tau}_{out}), x_{in}.\pi)$
    yield $(p_b, \tilde{e}_b)$
  // Assemble the code of these branches via a decision tree.
  return $\textsc{Destruct}_\Delta(x_{out}, \widehat{\tau}_{in}, B)$

Algorithm 10: Version of Seek adapted for a less painful proof.

To prepare for the main simulation proof, we redefine our compilation algorithms so that they emit code whose execution order is easily simulated by memory-level evaluation. For the purposes of our proofs, we discard the Wrap function and other mechanisms to emit recursive code. Extending our results and proofs to capture recursive code emission is left as future work [2]. The rest of our algorithms is modified in the following ways:

- we added tokens at appropriate places within the emitted target code to synchronize with memory-level source evaluation;
- we explicitly unroll type variables in memory layouts before exploring them further, and synchronize this with ETypeVar source evaluation steps;
- we split the Refine step of Rebuild into two parts: we first reify the shape of the unspecialized memory layout – i.e., memory structures common to all branches, then refine it into a specialized shape. Indeed, refining to the specialized shape requires that all splits have been processed, which is done through an ESplit step in source evaluation. We must therefore separate these two stages of shape refinement.

[2] Or to the reader, should they feel adventurous.

Data: Δ the type variable environment
Data: n input descriptions $(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i)$ for free variables in e
Data: x_{out} the destination location
Data: e the source expression
Result: Target expression $\tilde{e}$ storing its result in x_{out}
function $\textsc{Compile}_\Delta\,(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, x_{out}, e)$**:**
  cases e **:**
    case $(x_i.\pi : \tau \text{ as } \widehat{\tau})$ **:**
      return $\textsc{Seek}\,((x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i), (x_{out} : \tau \text{ as } \widehat{\tau}), \pi)$
    case $(u : \tau \text{ as } \widehat{\tau})$ **:**
      return $\textsc{Rebuild}\,(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau \text{ as } \widehat{\tau}), u)$
    case *let* $x : (\tau \text{ as } \widehat{\tau}) = e_0$ *in* e_1 **:**
      $\tilde{e}_0 \leftarrow \textsc{Compile}_\Delta\,(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, x, e_0)$
      $\tilde{e}_1 \leftarrow \textsc{Compile}_\Delta\,(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\} \cup \{(x \triangleleft \_ : \tau \text{ as } \widehat{\tau})\}, x_{out}, e_1)$
      return `let out` $x =$ `alloc`$(|\widehat{\tau}|)$; $\tilde{e}_0$; `freeze`(x)[*ELetBind-token*]; $\tilde{e}_1$
    case $f(x_i)$ **:**
      return `call` $f(x_i, x_{out})$[*EFunCall-token*]; `success`
    case *match*$(x_i)\,\{p_j \rightarrow e_j \mid 0 \leqslant j < N\}$ **:**
      for $i' \in \{0, \ldots, n-1\} \setminus \{i\}$ **do** $p'_{i'} \leftarrow p_{i'}$
      for $j \in \{0, \ldots, N-1\}$ **do**
        $p'_i \leftarrow p_j$
        $\tilde{e}_j \leftarrow \textsc{Compile}_\Delta\,(\{(x_{i'} \triangleleft p'_{i'} : \tau_{i'} \text{ as } \widehat{\tau}_{i'}) \mid 0 \leqslant i' < n\}, x_{out}, e_j)$
      return $\textsc{Destruct}_\Delta\,((x_i : \widehat{\tau}_i), \{(p_j, \textit{EMatch-token}; \tilde{e}_j) \mid 0 \leqslant j < N\})$

Algorithm 11: Version of Compile adapted for a less painful proof.

5.7.3 Simulation Relation on Environments

We now define a family of relations which synchronize evaluation environments of (source) memory expressions and target programs. They are defined in Fig. 5.21 for function environments ($\tilde{\mathcal{R}}_f$), input location environments ($\tilde{\mathcal{R}}_{in}$) and output location environments ($\tilde{\mathcal{R}}_{out}$). These three relations ensure that bindings are synchronised on both sides and that memory values bound in various environments have the same shape. Note that, for input locations, it proceeds by induction on call stacks, to find the right stack segment which contains the considered binding. We also add extra information in the source typing environment Γ to synchronise it with Rebuild and Seek calls, by remembering the current pattern of each input argument x_i. More precisely, each variable symbol x is bound to a triplet of the form $(p : \tau \text{ as } \widehat{\tau})$ in Γ.

$$\frac{\begin{array}{l}(f \mapsto \lambda x_{in}.e) \in \Sigma \qquad (f \mapsto \lambda x_{in} x_{out}.\tilde{e}) \in \tilde{\Sigma} \qquad \Delta, \Gamma \cup \{(x_{in} \triangleleft \_ : \tau_{in} \text{ as } \widehat{\tau}_{in})\} \vdash e : \tau_{out} \text{ as } \widehat{\tau}_{out} \\ \forall \varsigma_{src}, \forall \widehat{\nu}_{src}, \forall \varsigma_{tgt}, \forall \widehat{\nu}_{tgt}, \forall \rho, \forall \tilde{o}_{out}, \forall \varsigma_{tgt}, \forall a_{in}, \forall \widehat{\pi}_{in}, \\ \big(\Delta, \varsigma_{src} \vdash \widehat{\nu}_{src} : \widehat{\tau}_{in} \wedge \mathbf{shape\_of}_{\varsigma_{src}}(\widehat{\nu}_{src}) = \mathbf{shape\_of}_{\varsigma_{tgt}}(\widehat{\nu}_{tgt}) \\ \wedge \widehat{\mathbf{focus}}_{\varsigma_{tgt}}\,(\widehat{\pi}_{in}, \varsigma_{tgt}(a_{in})) = \widehat{\nu}_{tgt} \wedge \widehat{\mathbf{focus}}_{\varsigma_{tgt}}\,(\widehat{\pi}_{out}, \varsigma_{tgt}(a_{out})) = \_{|\widehat{\tau}_{out}|}\big) \\ \Rightarrow \Delta, x_{out} \vdash (\Gamma \cup \{(x_{in} \triangleleft \_ : \tau_{in} \text{ as } \widehat{\tau}_{in})\}, \{x_{in} \mapsto \widehat{\nu}_{src}\}, \varsigma_{src}, e)\ \tilde{\mathcal{R}}\ (\rho, \{x_{in} \mapsto a_{in}.\widehat{\pi}_{in}\}, \{x_{out} \mapsto a_{out}.\widehat{\pi}_{out}\}, \varsigma_{tgt}, \tilde{e})\end{array}}{\Delta, f \vdash \Gamma, \Sigma\ \tilde{\mathcal{R}}_f\ \tilde{\Sigma}}$$

$$\frac{\forall (f : (\tau_{in} \text{ as } \widehat{\tau}_{in}) \rightarrow (\tau_{out} \text{ as } \widehat{\tau}_{out})) \in \Gamma, \quad \Delta, f \vdash \Gamma, \Sigma\ \tilde{\mathcal{R}}\ \tilde{\Sigma}}{\Delta \vdash \Gamma, \Sigma\ \tilde{\mathcal{R}}_f\ \tilde{\Sigma}}$$

Figure 5.21: Relation $\tilde{\mathcal{R}}_f$ between function environments.

$$\frac{(x \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{\text{in}} \qquad \textbf{shape\_of}_{\varsigma_{\text{src}}}(\widehat{v}) = \textbf{shape\_of}_{\varsigma_{\text{tgt}}}\left(\widehat{\textbf{focus}}_{\varsigma_{\text{tgt}}}(\widehat{\pi}, \varsigma_{\text{tgt}}(a))\right)}{\Delta \vdash (x, \widehat{v}, \varsigma_{\text{src}})\ \tilde{\mathcal{R}}_{\text{in}}(\rho, \tilde{\sigma}_{\text{in}}, \varsigma_{\text{tgt}})}$$

$$\frac{\Delta \vdash (x, \widehat{v}, \varsigma_{\text{src}})\ \tilde{\mathcal{R}}_{\text{in}}(\rho, \tilde{\sigma}'_{\text{in}}, \varsigma_{\text{tgt}})}{\Delta \vdash (x, \widehat{v}, \varsigma_{\text{src}})\ \tilde{\mathcal{R}}_{\text{in}}\left((\tilde{\sigma}'_{\text{in}}, \tilde{\sigma}_{\text{out}}, \tilde{e}) :: \rho, \tilde{\sigma}_{\text{in}}, \varsigma_{\text{tgt}}\right)} \qquad \frac{\forall (x \mapsto \widehat{v}) \in \widehat{\sigma}, \Delta \vdash (x, \widehat{v}, \varsigma_{\text{src}})\ \tilde{\mathcal{R}}_{\text{in}}(\rho, \tilde{\sigma}_{\text{in}}, \varsigma_{\text{tgt}})}{\Delta \vdash (\widehat{\sigma}, \varsigma_{\text{src}})\ \tilde{\mathcal{R}}_{\text{in}}(\rho, \tilde{\sigma}_{\text{in}}, \varsigma_{\text{tgt}})}$$

Figure 5.21: Relation $\tilde{\mathcal{R}}_{\text{in}}$ between source and target input environments.

$$\frac{\widehat{e} \notin \widehat{\mathit{ValuExprs}} \qquad \Delta, \Gamma, \varsigma_{\text{src}} \vdash \widehat{e} : \widehat{\tau} \qquad (x_{\text{out}} \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{\text{out}} \qquad \widehat{\textbf{focus}}_{\varsigma_{\text{tgt}}}(\widehat{\pi}, \varsigma_{\text{tgt}}(a)) = \_|\widehat{\tau}|}{\Delta, x_{\text{out}} \vdash (\Gamma, \varsigma_{\text{src}}, \widehat{e})\ \tilde{\mathcal{R}}_{\text{out}}(\tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}})}$$

$$\frac{(x_{\text{out}} \mapsto a.\widehat{\pi}) \in \tilde{\sigma}_{\text{out}} \qquad \textbf{shape\_of}_{\varsigma_{\text{tgt}}}\left(\widehat{\textbf{focus}}_{\varsigma_{\text{tgt}}}(\widehat{\pi}, \varsigma_{\text{tgt}}(a))\right) = \textbf{shape\_of}_{\varsigma_{\text{src}}}(\widehat{u})}{\Delta, x_{\text{out}} \vdash (\Gamma, \varsigma_{\text{src}}, \widehat{u})\ \tilde{\mathcal{R}}_{\text{out}}(\tilde{\sigma}_{\text{out}}, \varsigma_{\text{tgt}})}$$

Figure 5.21: Relation $\tilde{\mathcal{R}}_{\text{out}}$ between source (memory) expressions and target output environments.

5.7.4 Seek and Rebuild

Before defining the simulation relation between full evaluation states, let us define some helper functions. We define the sequence of a list (or set if the order is arbitrary) of target expressions as:

$$\textbf{concat}(\{\tilde{e}_0, \ldots, \tilde{e}_{n-1}\}) \triangleq \tilde{e}_0; \ \ldots; \ \tilde{e}_{n-1}$$

Given Δ, ς and $\widehat{u}$ such that $\vDash \Delta, \varsigma \vdash \widehat{u} :$, we define notational shortcuts in Fig. 5.22 which establish a correspondence between the source memory valuexpression $\widehat{u}$ and parts of the target program emitted by Rebuild.

$$\text{consts}(\Delta, \varsigma, \widehat{u}) = \left\{ (\widehat{\pi}, \widehat{\tau}) \;\middle|\; \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{u}) = (u : \tau \text{ as } \widehat{\tau}) \right\}$$

$$\widehat{p}_{\text{consts}}(\Delta, \varsigma, \widehat{u}) = \textbf{shape\_of}_{\varsigma}(\widehat{u})\left[.\widehat{\pi} \leftarrow \textbf{shape\_of}_{\Delta}(\widehat{\tau}) \;\middle|\; (\widehat{\pi}, \widehat{\tau}) \in \text{consts}(\Delta, \varsigma, \widehat{u}) \right]$$

$$\text{splits}(\Delta, \varsigma, \widehat{u}) = \left\{ (\widehat{\pi}_0.\widehat{\pi}', \widehat{\tau}') \;\middle|\; \begin{array}{cc} & \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}_0, \widehat{u}) = (u : \tau \text{ as } \widehat{\tau}) \\ \wedge & \widehat{\textbf{focus}}_{\Delta}\,(\widehat{\pi}', \widehat{\tau}) = \text{split}\,(\dots) \\ \wedge & \exists!(p, \widehat{\tau}') \in \widehat{\tau}/\_, \Delta, \Gamma \vdash u : \tau/p \end{array} \right\}$$

$$\widehat{p}_{\text{splits}}(\Delta, \varsigma, \widehat{u}) = \widehat{p}_{\text{const}}(\Delta, \varsigma, \widehat{u})\left[.\widehat{\pi} \leftarrow \textbf{shape\_of}_{\Delta}(\widehat{\tau}) \;\middle|\; (\widehat{\pi}, \widehat{\tau}) \in \text{splits}(\Delta, \widehat{S}) \right]$$

$$\widehat{\Pi}_{\text{words}}(\Delta, \varsigma, \widehat{u}) = \left\{ \widehat{\pi}.\widehat{\pi}' \;\middle|\; \begin{array}{cc} & \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{u}) = (u : \tau \text{ as } \widehat{\tau}) \\ \wedge & \exists!(p, \widehat{\tau}') \in \widehat{\tau}/\_, \Delta, \Gamma \vdash u : \tau/p \\ \wedge & \widehat{\textbf{focus}}_{\Delta}\,(\widehat{\pi}', \widehat{\tau}') = \__{\ell} \end{array} \right\}$$

$$\text{frags}(\Delta, \varsigma, \widehat{u}) = \left\{ (\widehat{\pi}.\widehat{\pi}_f, \textbf{focus}\,(\pi_f, u)\,, \textbf{focus}\,(\pi_f, \tau)\,, \widehat{\tau}_f) \;\middle|\; \begin{array}{cc} & \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{u}) = (u : \tau \text{ as } \widehat{\tau}) \\ \wedge & \exists!(p, \widehat{\tau}') \in \widehat{\tau}/\_, \Delta, \Gamma \vdash u : \tau/p \\ \wedge & (\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \textbf{shatter}_{\Delta}(\widehat{\tau}') \end{array} \right\}$$

$$\widehat{\Pi}_{\text{addrs}}(\Delta, \varsigma, \widehat{u}) = \left\{ \widehat{\pi}.\widehat{\pi}' \;\middle|\; \begin{array}{cc} & \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{u}) = (u : \tau \text{ as } \widehat{\tau}) \\ \wedge & \exists!(p, \widehat{\tau}') \in \widehat{\tau}/\_, \Delta, \Gamma \vdash u : \tau/p \\ \wedge & \widehat{\textbf{focus}}_{\Delta}\,(\widehat{\pi}', \widehat{\tau}) = \&_{\ell}\,(\widehat{\tau}') \end{array} \right\} \cup \left\{ \widehat{\pi} \;\middle|\; \widehat{\textbf{focus}}_{\varsigma}\,(\widehat{\pi}, \widehat{u}) = \&_{\ell}\,(\widehat{u}') \right\}$$

Figure 5.22: Helper functions mirroring intermediate expressions emitted by Rebuild.

In order to exactly match $\hookrightarrow$ reduction steps, we should do the same for type variables that occur in pivots' memory types; for simplicity, we omit this and assume that type variables only occur in source-level pivots and in fragments.

We can now define our actual simulation relation $\tilde{\mathcal{R}}$ between source and target evaluation environments, in Fig. 5.23. It consists of five rules corresponding to the toplevel target code emitted by Compile; one rule IRRAtomFusion which allows the recombination of atoms destroyed by Seek; and three rules corresponding to intermediate execution stages of the target code emitted by Rebuild.

IRRLetBind

$$\frac{\Delta, x \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{e})\ \tilde{\mathcal{R}}\ \tilde{e} \qquad \tilde{e}' = \textsc{Compile}_{\Delta}\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\} \cup \{(x \triangleleft \_ : \tau \text{ as } \widehat{\tau})\}, x_{out}, e\right)}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \mathsf{let}\ x : \tau \text{ as } \widehat{\tau} = \widehat{e}\ \mathsf{in}\ e)\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}; \mathtt{freeze}(x)[\textsc{ELetBind}\text{-token}]; \tilde{e}')}$$

IRRStackSuccess

$$\frac{\tilde{\sigma}'_{out}(x'_{out}) = \tilde{\sigma}_{out}(x_{out}) \qquad \Delta, \{x_0, \ldots, x_{n-1}\}, x'_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{e})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}'_{out}, \tilde{e})}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{e})\ \tilde{\mathcal{R}}\ \left((\tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \mathtt{success}) :: \rho, \tilde{\sigma}'_{out}, \tilde{e}\right)}$$

IRRCompile

$$\frac{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad \Delta, \Gamma \vdash e : \tau \text{ as } \widehat{\tau}}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, e)\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \textsc{Compile}_{\Delta}\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau \text{ as } \widehat{\tau}), e\right))}$$

IRRSeek

$$\frac{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad \mathbf{focus}_{\Delta}(\pi, \tau_i) = \tau \qquad \Delta \vDash \widehat{\tau} \qquad \mathbf{agree}_{\Delta}(\tau, \widehat{\tau})}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, (x_i.\pi : \tau \text{ as } \widehat{\tau}))\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \textsc{Seek}_{\Delta}\left((x \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i), (x_{out} : \tau \text{ as } \widehat{\tau}), \pi\right))}$$

IRRRebuild

$$\frac{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad \Delta, \Gamma \vdash u : \tau \qquad \Delta \vDash \widehat{\tau} \qquad \mathbf{agree}_{\Delta}(\tau, \widehat{\tau})}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, (u : \tau \text{ as } \widehat{\tau}))\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \textsc{Rebuild}_{\Delta}\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau \text{ as } \widehat{\tau}), u\right))}$$

Figure 5.23: Simulation relation $\tilde{\mathcal{R}}$ between memory and target expressions.

IRRAtomFusion

$$\frac{\begin{array}{c}\Delta, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}) \\ \exists o_0, \ell_0, \ldots, o_{n-1}, \ell_{n-1}, \mathbf{shape\_of}_{\varsigma}(\widehat{u}) = \__{\ell} \bowtie_{0 \leqslant i < n} [o_i : \ell_i] : \__{\ell_i} \\ o_0 = 0 \qquad o_{i+1} = o_i + \ell_i \qquad o_{n-1} + \ell n - 1 = \ell\end{array}}{\Delta, \{x_0, \ldots, x_{N-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e};\ \mathtt{cast}\ x_{out}\ \mathtt{to}\ I_{\ell};\ \mathtt{success})}$$

Figure 5.23: Simulation relation for memory valuexpressions: Seek code.

IRRRebuildConsts

$$\frac{\begin{array}{c} x_{out}, \widehat{p}_{consts}(\Delta, \varsigma, \widehat{u}) \vdash (\varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{const}(\tilde{\sigma}_{out}, \tilde{e}_{consts}) \\ \tilde{e}_{splits} = \textbf{concat}\left(\left\{\text{ESplit-token} \;\middle|\; (\widehat{\pi}, \widehat{\tau}) \in splits(\Delta, \varsigma, \widehat{u})\right\}\right) \\ \tilde{e}'_{consts} = \textsc{Refine}\left(x_{out}, \widehat{p}_{const}(\Delta, \varsigma, \widehat{u}), \widehat{p}_{splits}(\Delta, \varsigma, \widehat{u})\right) \\ \tilde{e}_{words} = \textbf{concat}\left(\left\{\text{EWord-token} \mid \widehat{\pi} \in \widehat{\Pi}_{words}(\Delta, \varsigma, \widehat{u})\right\}\right) \\ \Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{frags}(\rho, \tilde{\sigma}_{out}, \tilde{e}_{frags}) \end{array}}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}_{consts}; \tilde{e}_{splits}; \tilde{e}'_{consts}; \tilde{e}_{words}; \tilde{e}_{frags})}$$

IRRRebuildSplits

$$\frac{\begin{array}{c} \tilde{e}_{splits} = \textbf{concat}\left(\left\{\text{ESplit-token} \;\middle|\; (\widehat{\pi}, \widehat{\tau}) \in splits(\Delta, \varsigma, \widehat{u})\right\}\right) \\ x_{out}, \widehat{p}_{splits}(\Delta, \varsigma, \widehat{u}) \vdash (\varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{const}(\tilde{\sigma}_{out}, \tilde{e}_{consts}) \\ \tilde{e}_{words} = \textbf{concat}\left(\left\{\text{EWord-token} \mid \widehat{\pi} \in \widehat{\Pi}_{words}(\Delta, \varsigma, \widehat{u})\right\}\right) \\ \Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{frags}(\rho, \tilde{\sigma}_{out}, \tilde{e}_{frags}) \end{array}}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}_{splits}; \tilde{e}_{consts}; \tilde{e}_{words}; \tilde{e}_{frags})}$$

IRRRebuildFrags

$$\frac{\widehat{\textbf{focus}}\ (\widehat{\pi}, C) = \square \qquad \begin{array}{c} \tilde{\sigma}_{out}(x'_{out}) = \tilde{\sigma}_{out}(x_{out}).\widehat{\pi} \qquad \Delta, \{x_0, \ldots, x_{n-1}\}, x'_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}) \\ \Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, C[\_{|\widehat{u}|}])\ \tilde{\mathcal{R}}_{frags}(\rho, \tilde{\sigma}_{out}, \tilde{e}_{frags}) \end{array}}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, C\,[\widehat{u}])\ \tilde{\mathcal{R}}\ (\rho, \tilde{\sigma}_{out}, \tilde{e}; \tilde{e}_{frags})}$$

Figure 5.23: Simulation relation for memory valuexpressions: Rebuild code.

The $\tilde{\mathcal{R}}_{frags}$ relation defined in Fig. 5.24 characterizes the portion of target code emitted by Rebuild which deals with non-constant parts of the memory type – i.e., $\tilde{e}_{frags}; \tilde{e}_{addrs}$. Its definition follows exactly the corresponding portions of the Rebuild algorithm.

$$\frac{\begin{array}{c} (x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad (x_{out} \mapsto a_{out}.\widehat{\pi}_{out}) \in \tilde{\sigma}_{out} \qquad frags(\Delta, \varsigma_{src}, \widehat{u}) = \{(\widehat{\pi}_f, u_f, \tau_f, \widehat{\tau}_f) \mid 0 \leqslant f < N\} \\ \forall (\widehat{\pi}, \widehat{\tau}) \in consts(\Delta, \varsigma_{src}, \widehat{u}), \exists f \in \{0, \ldots, N-1\}, \widehat{\pi} = \widehat{\pi}_f \\ \tilde{e}_f = \begin{cases} \textsc{Seek}_\Delta\left((x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i), (x_f : \tau_f \text{ as } \widehat{\tau}_f), \pi\right) & \text{if } u_f = x_i.\pi \\ \textsc{Rebuild}_\Delta\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_f : \tau_f \text{ as } \widehat{\tau}_f), u_f\right) & \text{otherwise} \end{cases} \\ \Delta, \{x_0, \ldots, x_{n-1}\}, x_f \vdash (\Gamma, \widehat{\sigma}, \varsigma, (u_f : \tau_f \text{ as } \widehat{\tau}_f))\ \tilde{\mathcal{R}}\ \left(\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out} \cup \{x_f \mapsto a_{out}.\widehat{\pi}_{out}.\widehat{\pi}_f\}, \varsigma_{tgt}, \tilde{e}_f\right) \\ \tilde{e}_{tf} = \begin{cases} \texttt{success} & \text{if } \widehat{\textbf{focus}}_\varsigma\ (\widehat{\pi}_f, \widehat{u}) = (u : \tau \text{ as } I_\ell) \\ \text{EFragment-token}; \texttt{success} & \text{otherwise} \end{cases} \\ \tilde{e}_{frags} = \textbf{concat}\left(\{\texttt{let out } x_f = x_{out}.\widehat{\pi}_f;\ \tilde{e}_{tf};\ \tilde{e}_f \mid 0 \leqslant f < N\}\right) \\ \tilde{e}_{addrs} = \textbf{concat}\left(\left\{\text{EAddress-token} \;\middle|\; \widehat{\pi} \in \widehat{\Pi}_{addrs}(\Delta, \varsigma, \widehat{u})\right\}\right) \end{array}}{\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma, \widehat{u})\ \tilde{\mathcal{R}}_{frags}(\rho, \tilde{\sigma}_{out}, \tilde{e}_{frags}; \tilde{e}_{addrs})}$$

Figure 5.24: The $\tilde{\mathcal{R}}_{frags}$ relation

5.7.5 Statement of Simulation

Theorem 5.1 ($\tilde{\mathcal{R}}$ is a weak simulation). *Let* Δ, Σ, $\widehat{S} = (\Gamma, \widehat{\sigma}, \varsigma_{src}, \widehat{e})$, $\tilde{\Sigma}$, $\tilde{S} = (\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e})$, x_{out} *and* $x_0, \ldots, x_{n-1}$ *such that*

$$\vDash \Delta \vdash \Gamma \qquad\qquad \Delta, \varsigma \vdash \widehat{\sigma} : \Gamma$$

$$\forall i \in \{0, \ldots, n-1\}, (x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \Rightarrow \left(\Delta \vdash p_i : \tau_i \wedge \exists (p'_i, \widehat{p}_i) \in \textbf{pat2mem}_\Delta(p_i, \widehat{\tau}_i), \varsigma \vdash \widehat{p}_i \blacktriangleright \widehat{\sigma}(x_i)\right)$$

$$\Delta \vdash (\Gamma, \Sigma)\ \tilde{\mathcal{R}}_f\ \tilde{\Sigma} \qquad \Delta \vdash (\widehat{\sigma}, \varsigma_{src})\ \tilde{\mathcal{R}}_{in}\ (\rho, \tilde{\sigma}_{in}, \varsigma_{tgt}) \qquad \Delta, x_{out} \vdash (\Gamma, \varsigma_{src}, \widehat{e})\ \tilde{\mathcal{R}}_{out}\ (\tilde{\sigma}_{out}, \varsigma_{tgt})$$

If

$$\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash \widehat{S}\ \tilde{\mathcal{R}}\ \tilde{S}$$

then one of the three following conditions holds:

- *Both* $\widehat{S}$ *and* $\tilde{S}$ *are in normal form w.r.t.* $\not\mapsto$ *and* $\not\rightsquigarrow$ *respectively.*
- *There exists* $\tilde{S}'$ *such that*

$$\tilde{\Sigma} \vdash \tilde{S} \rightsquigarrow_{\succeq} \tilde{S}' \qquad\qquad \Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash \widehat{S}\ \tilde{\mathcal{R}}\ \tilde{S}'$$

- *There exist a memory-level evaluation step label* L, $\widehat{S}'$, $\tilde{S}'$, $\tilde{S}''$ *and a finite number of steps* m *such that*

$$\Delta, \Sigma \vdash \widehat{S} \mapsto_L \widehat{S}' \qquad \tilde{\Sigma} \vdash \tilde{S} \rightsquigarrow^m_{\succeq} \tilde{S}' \rightsquigarrow_L \tilde{S}'' \qquad \Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash \widehat{S}'\ \tilde{\mathcal{R}}\ \tilde{S}''$$

Proof. By induction on $\tilde{\mathcal{R}}$.

IRRCompile: we have

$$\widehat{e} = e \in \mathit{Exprs} \qquad \tilde{e} = \textsc{Compile}_\Delta\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, x_{out}, e\right)$$

and proceed by case analysis on e and $\tilde{e}$. If e is a pivot, it is handled by the IRRSeek and IRRRebuild cases below. The remaining Compile cases are let-binding, function call and pattern matching expressions. Here, we focus on the match case: there exists $i \in \{0, \ldots, n-1\}$ such that

$$e = \mathsf{match}(x_i)\left\{ p_j \ \rightarrow\ e_j \ \middle|\ 0 \leqslant j < N \right\}$$

$$\tilde{e} = \textsc{Destruct}_\Delta\left((x_i : \widehat{\tau}_i), \left\{(p_j, \text{EMatch-token}; \tilde{e}_j) \ \middle|\ 0 \leqslant j < N\right\}\right)$$

where for each branch $j \in \{0, \ldots, N-1\}$,

$$\tilde{e}_j = \textsc{Compile}_\Delta\left(\{(x_{i'} \triangleleft p'_{i'} : \tau_{i'} \text{ as } \widehat{\tau}_{i'}) \mid 0 \leqslant i' < n\}, x_{out}, e_j\right) \qquad p'_{i'} = \begin{cases} p_j & \text{if } i' = i \\ p_{i'} & \text{otherwise} \end{cases}$$

Using Theorem 3.2, there exists at least one branch $j \in \{0, \ldots, N-1\}$ such that

$$\exists (p', \widehat{p}) \in \textbf{pat2mem}_\Delta(p_j, \widehat{\tau}_i), \varsigma_{src} \vdash \widehat{p} \blacktriangleright \widehat{\sigma}(x_i)$$

We pick the smallest such j. Using Theorem 4.1 and the definition of Destruct, there exists a finite number of steps m such that

$$\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e} \rightsquigarrow^m_{\succeq} \underbrace{\rho, \tilde{\sigma}'_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \text{EMatch-token}; \tilde{e}_j}_{\tilde{S}'}$$

with $\tilde{\sigma}_{in} \subseteq \tilde{\sigma}'_{in}$ (which preserves $\tilde{\mathcal{R}}_{in}$). At this point, both programs ($\widehat{S}$ and $\tilde{S}'$) go through an EMatch evaluation step and yield the following states:

$$\widehat{S}' = (\Gamma, \widehat{\sigma}, \varsigma_{src}, e_j) \qquad\qquad \tilde{S}'' = (\rho, \tilde{\sigma}'_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e}_j)$$

and we conclude using the IRRCompile rule.

IRRSeek: there exists $i \in \{0, \ldots, n-1\}$ such that we have

$$(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad \textbf{focus}_\Delta(\pi, \tau_i) = \tau \qquad \Delta \vDash \widehat{\tau} \qquad \textbf{agree}_\Delta(\tau, \widehat{\tau}) \qquad \widehat{e} = (x_i.\pi : \tau \text{ as } \widehat{\tau})$$

$$\tilde{e} = \textsc{Seek}_\Delta\left((x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i), (x_{out} : \tau \text{ as } \widehat{\tau}), \pi\right)$$

Here, we only consider the case where $\widehat{\tau}$ is not a type variable, either $\pi \neq \varepsilon$ or $\widehat{\tau}_i \neq \widehat{\tau}$, and Seek finds a fragment within the input value on which to focus. The target program $\tilde{e}$ first goes through the decision tree emitted by Destruct in a finite number m of $\lesssim$-transitions, reaching the target expression $\tilde{e}_b$ of the corresponding input branch $(p_b, \tau_b, \widehat{\tau}_b, \text{frags}_b) \in \textbf{Explore}_\Delta(p_i, \tau_i, \widehat{\tau}_i)$. There exists a fragment $(\widehat{\pi}_f \mapsto \pi_f \text{ as } \widehat{\tau}_f) \in \text{frags}_b$ and a suffix path π' such that $\pi = \pi_f.\pi'$, and we have

$$\tilde{e}_b = \texttt{let in } x_f = x_i.\widehat{\pi}_f; \text{EVarFocus-token}; \tilde{e}_f$$

where x_f is a fresh symbol and

$$\tilde{e}_f = \textsc{Seek}_\Delta\left((x_f \triangleleft p_f : \tau_f \text{ as } \widehat{\tau}_f), (x_{out} : \tau \text{ as } \widehat{\tau}), \pi'\right)$$

with

$$\tau_f = \textbf{focus}_\Delta(\pi_f, \tau_b) \qquad\qquad p_f = \textbf{focus}(\pi_f, p_b)$$

From there, the target expression $\tilde{e}_b$ goes through a $\lesssim$-transition (IRESubInLoc rule) to execute the `let in` instruction. Both programs then go through a EVarFocus step: the target program consumes its token, and the source expression performs the following reduction step:

$$\Gamma, \widehat{\sigma}, \varsigma_{src}, (x_i.\pi : \tau \text{ as } \widehat{\tau}) \leadsto \Gamma \cup \{(x_f : \tau_f \text{ as } \widehat{\tau}_f)\}, \widehat{\sigma} \cup \{x_f \mapsto \widehat{\textbf{focus}}_{\varsigma_{src}}(\widehat{\pi}_f, \widehat{\sigma}(x_i))\}, \varsigma_{src}, (x_f.\pi' : \tau \text{ as } \widehat{\tau})$$

We conclude using the IRRSeek rule.

IRRRebuild: we have

$$\widehat{e} = (u : \tau \text{ as } \widehat{\tau}) \qquad (x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \in \Gamma \qquad \Delta, \Gamma \vdash u : \tau \qquad \Delta \vDash \widehat{\tau} \qquad \textbf{agree}_\Delta(\tau, \widehat{\tau})$$

$$\tilde{e} = \textsc{Rebuild}_\Delta\left(\{(x_i \triangleleft p_i : \tau_i \text{ as } \widehat{\tau}_i) \mid 0 \leqslant i < n\}, (x_{out} : \tau \text{ as } \widehat{\tau}), u\right)$$

Here, we ignore the type variable and primitive base cases. Let

$$\text{Pos}_{out} = \{(x.\pi, \pi') \mid \textbf{focus}(\pi', u) = x.\pi\}$$

$$p_{out} = \textbf{Remap}\left(\{x_i \mapsto p_i \mid 0 \leqslant i < n\}, u\,[x.\pi \mapsto \_], \text{Pos}_{out}\right)$$

The target program $\tilde{e}$ first goes through the decision tree emitted by Destruct in a finite number m of $\lesssim$-transitions, reaching the target expression $\tilde{e}_b$ of the corresponding output branch $(p_b, \tau_b, \widehat{\tau}_b, \text{frags}_b) \in \textbf{Explore}_\Delta(p_{out}, \tau, \widehat{\tau})$. It is immediate from their definitions that $\widehat{\Pi}_{words}$, $\widehat{\Pi}_{addrs}$, $\widehat{\Pi}_{splits}$ defined in the Rebuild algorithm are identical to those derived from the source state using helper functions in Fig. 5.22. It is also immediate from their definitions that the consistuent pieces of $\tilde{e}_b$ $\tilde{e}_{splits}$, $\tilde{e}'_{consts}$, $\tilde{e}_{words}$ are identical to those derived from the source state in the IRRRebuildConsts rule. The target expressions $\tilde{e}_{frags}$ and $\tilde{e}_{addrs}$ are defined in such a way that we have

$$\Delta, \{x_0, \ldots, x_{n-1}\}, x_{out} \vdash (\Gamma, \widehat{\sigma}, \varsigma_{src}, (u : \tau \text{ as } \widehat{\tau}))\ \tilde{\mathcal{R}}_{\text{frags}}(\rho, \tilde{\sigma}_{in}, \tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e}_{frags}; \tilde{e}_{addrs})$$

Finally, the first piece of $\tilde{e}_b$ is defined as follows:

$$\tilde{e}_{consts} = \textsc{Refine}\left(x_{out}, \_|\widehat{\tau}|, \textbf{shape\_of}_\Delta(\widehat{\tau})\right)$$

and according to the precondition on output values, we have

$$x_{out}, \_|\widehat{\tau}| \vdash (\varsigma_{src}, (u : \tau \text{ as } \widehat{\tau}))\ \tilde{\mathcal{R}}_{\text{const}}(\tilde{\sigma}_{out}, \varsigma_{tgt}, \tilde{e}_{consts})$$

We conclude with the IRRRebuildConsts rule using Lemma 5.1.

Other rules: immediate by induction or done similarly to previous cases.

□

5.7.6 Future work

We have shown that our compilation algorithms, if they succeed, emit target code whose execution is simulated by the source program's memory-level evaluation. Ideally, we would also state and prove the following results:

- Given well-formed inputs, Compile and other compilation procedures succeed. It would probably involve a cumbersome, but not fundamentally difficult, induction on the source expression and types, similar to many of the proofs presented in Section 3.4.
- Termination (and correctness) of the memoized versions of Seek and Rebuild. The termination property would stem from the fact that input programs are finite. We would extend the existing correctness proofs with new base cases corresponding to hashmap hits.

5.8 Conclusion

In this chapter, we presented a unified compilation procedure for the whole Ribbit language. The Ribbitulus combines a small high-level language, manipulating immutable ADT values with pattern matching, accessors and data constructors, with a much lower-level memory specification language. Although both of these components are relatively easy to compile on their own, their combination creates new compilation problems.

In order to define a complete compilation scheme for the Ribbitulus, we considered each language construct separately, yielding a collection of compilation procedures dedicated to specific aspects of our language. We provide a toplevel compilation interface which handles the full language by combining these smaller procedures together.

Two aspects of our language, namely pattern matching and pivot expressions, stand out as their compilation is significantly complicated by custom memory layouts. The compilation procedures handling these two language constructs are therefore driven by the exploration of their memory types, following their structure to emit appropriate target code.

Our target is an intermediate representation in Destination-Passing Style which provides the necessary tools for both of these aspects: decision nodes are used as the target of pattern matching compilation, while explicit memory allocations, reads and writes support the precise manipulation of memory contents required by pivot expressions.

We were able to consider pattern matching in isolation: in Chapter 4, we provided a compilation approach solely for the "recognition" aspect (i.e., excluding variable bindings) of pattern matching. It took custom memory layouts into account by lowering high-level patterns to the memory-level language.

We were not able to isolate the compilation of pivot expressions as much: indeed, the combination of data constructors and accessors causes interaction between arbitrary memory layouts. Since the Ribbit language provides enough flexibility to specify any combination of memory layouts, as long as they agree with the considered high-level data type, this situation may cause seemingly trivial data constructors/accessors to require complex recursive target code.

We established the correctness of our compilation scheme w.r.t. the source language by proving that each of its components was correct w.r.t. the corresponding Ribbitulus fragment.

Part III

The Ribbit Implementation

In the first two parts of this thesis, we presented the Ribbit language and a full formal compilation approach for this language. This compilation approach included some fairly complex algorithms, for which it is desirable to have some form of practical validation. In this final part, we detail our implementation of these algorithms in the Ribbit prototype compiler. Chapter 6 describes the compiler itself and details how it implements complex procedures such as our mutually recursive and memoized Rebuild and Seek. This implementation allowed us to carry out a preliminary experimental evaluation of our pattern matching compilation approach, which we will present in Chapter 7.

Chapter 6

Ribbit compiler implementation

This chapter describes our prototype implementation of the Ribbit language and compiler. It is written in OCaml and available at `https://gitlab.inria.fr/ribbit/ribbit` under the MIT license. In addition to the source code, the Ribbit distribution includes various example input files, including the memory zoo exhibits from Chapter 2. A web interface is also available at `https://ribbit.gitlabpages.inria.fr/ribbit/` to experiment with the Ribbit language and with predefined examples interactively.

Our implementation can verify the validity of ADTs, memory layouts and programs, compile programs to our intermediate representation, show the obtained CFGs, and run them. In addition, it will verify the correctness of compiled code against a reference source interpreter.

Note that the input Ribbit syntax shown in this chapter differs slightly from that shown in exhibits in Chapter 2. Indeed, at the time of writing, the Ribbit implementation still uses a previous version of memory types with minor differences in how constant, pointer and composite words are modeled.

6.1 Running example: arithmetic expressions

Recall the arithmetic expression example from the Memory Zoo (Section 2.3). Listing 1 shows the contents of a Ribbit input file (extension `.rbt`) which defines the ADT `Exp` for arithmetic expressions with an optimized memory layout corresponding to `ExpOpt`, along with the `eval` function.

As mentioned before, there are slight differences between the idealized syntax of Chapter 2 and the concrete input syntax accepted by the current Ribbit prototype. In this program, they manifest themselves in the following ways:

- Both primitive (integer) types and unspecified word types are modeled with `wl`, which designates an `l`-bit "word" without specifying its contents. When it appears on the left-hand side of a bit range specification `with ..., wl` acts as an uninitialized word type (`_<l>` in idealized syntax) which is used to build a composite word type. When it occurs on its own, without bit ranges specifications, `wl` is interpreted as an integer encoding (`il` in idealized syntax).
- Memory words whose contents are set to a given constant `c` are expressed with a "whole-word" specification `with . : and` an unsized singleton type `(= c)` added onto an adequately sized word type `wl`, rather than with a constant word type `(c)<l>` as in Chapter 2.
- The number of bits which are unused due to address alignment is specified for each pointer type – for instance, `&<64, 2>(...)` indicates that the two lowest bits may be used to store extra data.
- All other syntactical differences are purely cosmetic: bit ranges are denoted `[o, l]` (rather than `[o:l]`), subterm paths in fragments are prefixed with `_`, and function call arguments are specified in a rather strange way – for instance, `f(x=v)` rather than simply `f(v)`.

```
type String = i512;
represented as (_ as w512)

enum Op { Plus, Mult }
represented as w8 with . : split . {
  | 0 from Plus => (= 0)
  | 1 from Mult => (= 1)
}

enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) }
represented as split .[0, 2] {
  | 0 from Bin(_, Bin|Var, Bin|Var) =>
    &<64, 2>({{ _.Bin.0 as Op, w56 with . = 0, _.Bin.1 as Exp, _.Bin.2 as Exp }})
  | 1 from Int => w64 with .[32, 32] : (_.Int as w32)
  | 2 from Var => &<64, 2>(_.Var as String)
  | 3 from Bin(_, Int, Bin|Var)|Bin(_, Bin|Var, Int)|Bin(_, Int, Int) =>
    &<64, 2>(split .1 {
      | 0 from Bin(_, Int, _) =>
        {{ _.Bin.0 as Op, w24, _.Bin.1.Int as w32, _.Bin.2 as Exp }}
      | 1 from Bin(_, Bin|V, Int) =>
        {{ _.Bin.0 as Op, w24, _.Bin.2.Int as w32, _.Bin.1 as Exp }}
    })
}

let v : Exp = Bin(Plus, Int(42), Var(0));

fn eval (e : Exp) -> Exp {
  match e {
    Int(_) | Var(_) => e,
    Bin(op, e1, e2) => match (eval(e=e1), eval(e=e2)) {
      (Int(n1), Int(n2)) => match op {
        Plus => Int(n1 + n2),
        Mult => Int(n1 * n2)
      },
      (e1', e2') => Bin(op, e1', e2')
    }
  }
}

let res : Exp = eval(e=v);
```

Listing 1: Optimized memory layout for arithmetic expressions and user implementation of `eval`.

6.2 Intermediate forms and interpreters

In this section, we provide an overview of the various intermediate forms that a Ribbit program goes through during its compilation. Many of these intermediate program representations have their own interpreter, which lets us check empirically that the semantics of a given program is preserved by each compilation pass. We will use the program on arithmetic expressions from Section 6.1 as our running example to illustrate each representation. All of them can be obtained at once by invocating `ribbit` with verbose, graph and debug options enabled, using the command `ribbit -m batch -ir -g -s -debug arithexprs.rbt`. Crucially, all intermediate forms shown in this section were taken straight from the compiler's output, without any modifications.

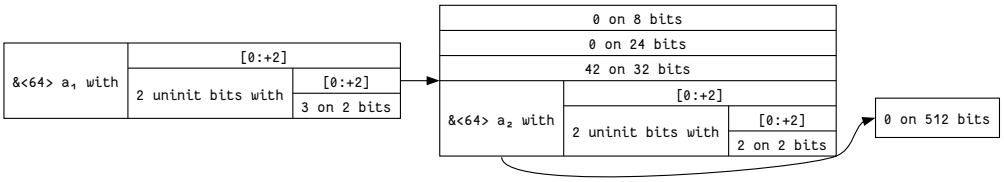
The Ribbit prototype first processes all type declarations before moving on to function and value

declarations. Let us illustrate this processing on the `String`, `Op` and `Exp` type declarations from the program shown in Listing 1. This step normalizes memory types given by programmers, or computes them based on a generic representation (as described in Section 2.6). The syntax is exactly the same as memory types, with some minor expansions (such as expanded or-patterns in split branches' provenances).

```
[Type declarations]:
enum Exp {Var(String), Int(i32), Bin(Op, Exp, Exp)}
represented as
split(_.[0:+2]) (
  0 from
    Bin(_, Bin(_), Bin(_)), Bin(_, Bin(_), Var(_)),
    Bin(_, Var(_), Bin(_)), Bin(_, Var(_), Var(_)) =>
    &<64,2>({{_.Bin.0 as Op, w56 w _:(= 0), _.Bin.1 as Exp, _.Bin.2 as Exp}})
       w _.[0:+2]:(= 0),
  1 from Int(_) => w64 w _.[0:+2]:(= 1) w _.[32:+32]:_.Int as w32,
  2 from Var(_) => &<64,2>(_.Var as String) w _.[0:+2]:(= 2),
  3 from
    Bin(_, Bin(_), Int(_)), Bin(_, Int(_), Bin(_)),
    Bin(_, Int(_), Int(_)), Bin(_, Int(_), Var(_)),
    Bin(_, Var(_), Int(_)) =>
    &<64,2>(split(_.1) (
              0 from Bin(_, Int(_), _) =>
                {{_.Bin.0 as Op,
                  w24 w _:(= 0),
                  _.Bin.1.Int as w32,
                  _.Bin.2 as Exp}},
              1 from Bin(_, Bin(_), Int(_)), Bin(_, Var(_), Int(_)) =>
                {{_.Bin.0 as Op,
                  w24 w _:(= 1),
                  _.Bin.2.Int as w32,
                  _.Bin.1 as Exp}}
            ))
       w _.[0:+2]:(= 3)
)

enum Op {Plus, Mult}
represented as
w8 w _:split(_) (
         0 from Plus(_) => (= 0),
         1 from Mult(_) => (= 1)
       )

type String = i512 represented as _ as w512
```

Listing 2: Ribbit output: Processing type declarations.

Let us now consider the value declaration from the program shown in Listing 1: `let v : Exp = Bin(Plus, Int(42), Var(0));`. Ribbit outputs its representation as a memory value in both textual and graphical form, as shown in Listing 3.




```
[Value Declaration]:
Compiling Bin(Plus, Int(42), Var(0))

[Value Declaration]:
Elaborated to Bin(Plus, Int(42), Var(0))

[Value Declaration]:
v = Bin(Plus, Int(42), Var(0)) = Bin(Plus, Int(42), Var(0))
represented as
 &<64>(a0) with [0:+2] = ?2 with [0:+2] = (3)2
 with a0 ->
        {{ (0)8;
           (0)24;
           (42)32;
           &<64>(a1) with [0:+2] = ?2 with [0:+2] = (2)2;
        }};
       a1 -> (0)512
```

Listing 3: Ribbit output: Processing the first value declaration.

In general, the Ribbit prototype processes such declarations with the following sequence of actions:

- typecheck and desugar the bound expression;
- prepare new identifiers and compute the shape of its memory type;
- compile the expression and optimize the resulting target expression;
- prepend an allocation instruction and append a freeze instruction to the resulting target expression;
- evaluate the desugared source expression as a purely high-level object (↪ then substitute all variables), yielding a high-level value;
- evaluate the compiled expression with the target interpreter, yielding a memory value and its accompanying store;
- convert the high-level value to a memory value;
- compare both memory values to check that both routes of evaluation yield the same result;
- if both succeed and yield equivalent memory values, add both high-level and memory values to the current value environment.

This means that, for every program, we check the correctness of Compile empirically. This has proven crucial in practice to converge towards a correct compilation algorithm.

Let us now consider the successive intermediate forms of a full-fledged function: `eval`. The Ribbit prototype first typechecks and normalizes its body to the explicitly typed A-normal form shown in Listing 4. This elaboration was already showcased in Section 2.3.

```
[Function declaration]:
Compiling eval(e: ) = match e {
Int(_) | Var(_) => e,
  Bin(op, e1, e2) => match (eval(
  e: e1), eval(e: e2)) {
  (Int(n1), Int(n2)) => match op {
  Plus => Int(n1 + n2),
    Mult => Int(n1 * n2),
  },
    (e1', e2') => Bin(op, e1', e2'),
  },
}

[Function declaration]:
Elaborated to eval(e: Exp) : Exp
= let x : Exp = e;
  match x {
    Int(_) => e,
    Var(_) => e,
    Bin(_, _, _) =>
      let x1 : Exp = x.Bin.1;
      let x2 : Exp = eval(e: x1);
      let x3 : Exp = x.Bin.2;
      let x4 : Exp = eval(e: x3);
      match x2, x4 {
        Int(_), Int(_) =>
          let x5 : Op = x.Bin.0;
          match x5 {
            Plus =>
              let x6 : i32 = x2.Int;
              let x7 : i32 = x4.Int;
              let x8 : i32 = ADD(x6, x7);
              Int(x8),
            Mult =>
              let x9 : i32 = x2.Int;
              let x10 : i32 = x4.Int;
              let x11 : i32 = MUL(x9, x10);
              Int(x11),
            },
        _, _ => Bin(x.Bin.0, x2, x4),
        },
    }
```

Listing 4: Ribbit output: Desugared `eval`.

Following our compilation algorithms, we then process the pattern matching and each right-hand side expression. The resulting CFG is given in Fig. 6.1. This CFG was already shown, with some minor optimizations applied, in Section 2.3.

Finally, we evaluate `eval(v)` in two ways: as a source expression using Ribbit's built-in interpreter for the source language, and as a target expression using its compiled version and Ribbit's built-in interpreter for the target language. Here, both evaluations yield equivalent results, and we obtain the output shown in Listing 5, which represents the result as a memory value. Note the use of `?2` to represent the 2 uninitialized padding bits.

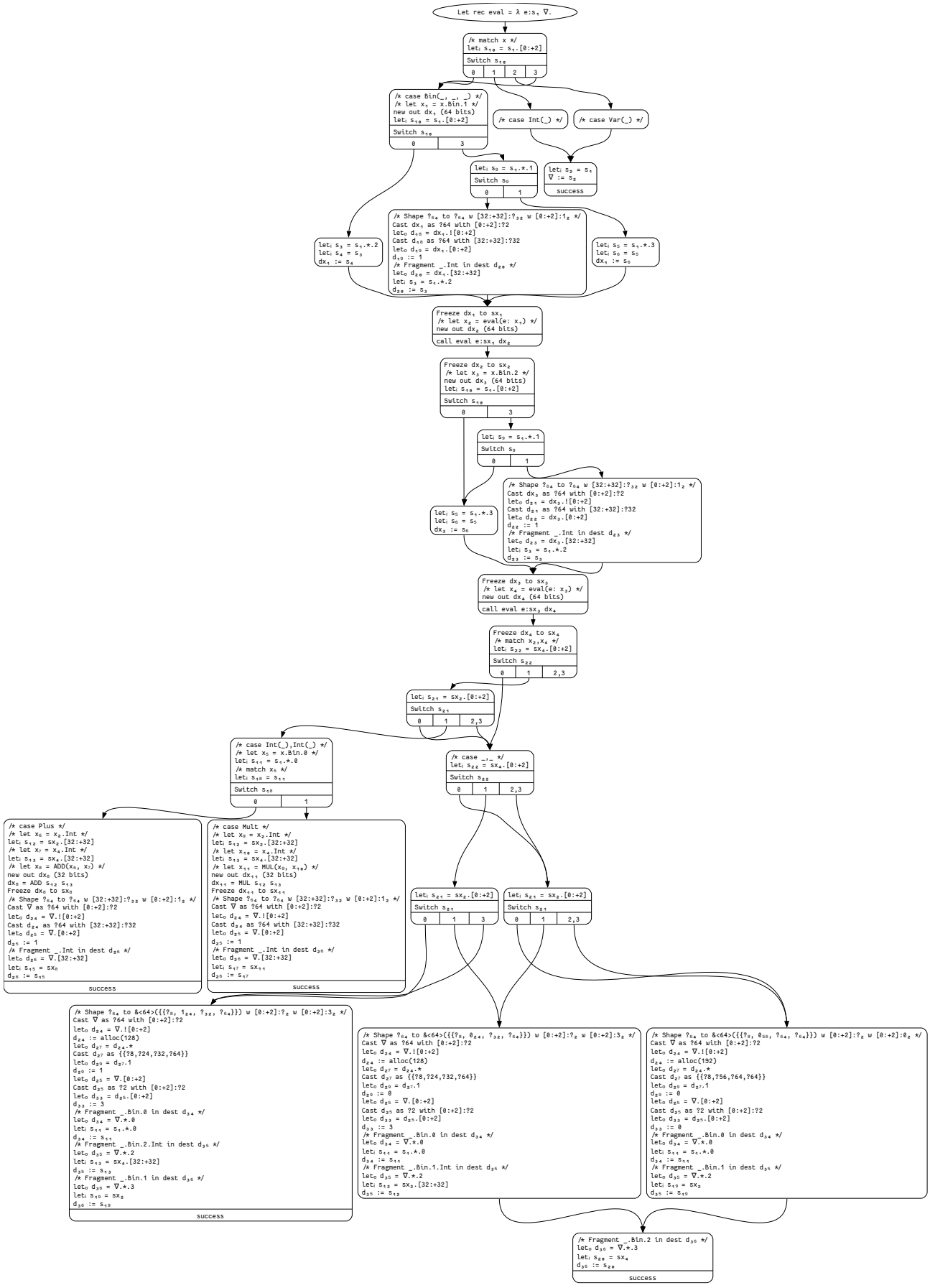
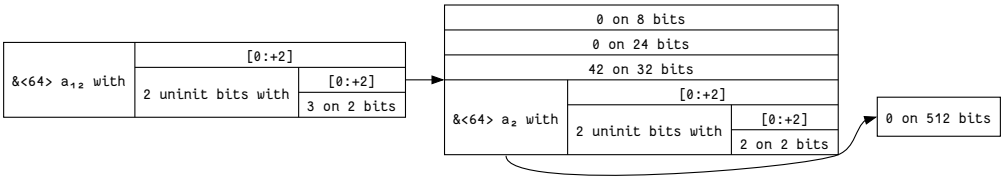


Figure 6.1: Ribbit output: target CFG for `eval`.




```
[Value Declaration]:
Compiling eval(e: v)

[Value Declaration]:
Elaborated to let x12 : Exp = v;
              eval(e: x12)

[Value Declaration]:
res = eval(e: v) = Bin(Plus, Int(42), Var(0))
represented as
 &<64>(a0) with [0:+2] = ?2 with [0:+2] = (3)2
 with a0 ->
        {{ (0)8;
           (0)24;
           (42)32;
           &<64>(a1) with [0:+2] = ?2 with [0:+2] = (2)2;
        }};
       a1 -> (0)512
```

Listing 5: Ribbit output: Evaluation of `eval(v)`.

6.3 Technical Features

In this section, we go over some specific technical aspects of the Ribbit implementation.

Hash-consing Classically, directed acyclic graphs (DAGs) with maximal sharing can be created from trees by using hash-consing (Filliâtre and Conchon 2006). For both termination and tractability, we apply these techniques pervasively in Ribbit, both on memory trees (described in Section 4.2) and on target IR expressions (described in Section 5.2).

For this purpose, we use the `Hashtbl` interface from OCaml standard library to provide a generic hash-consing-aware recursion scheme for trees. Listing 6 shows some parts of Ribbit's source code using this interface. The fixpoint operator `memo_rec` memoizes a given function `f` by hash-consing its input. We use it to define a sharing-aware fold over memory trees. This fold recovers maximal sharing, even when it was not originally present. We use this fold pervasively throughout pattern matching compilation.

```ocaml
(** Memoize a function on Memory Trees by hash-consing them. *)
let memo_rec f =
  let tbl = Hashtbl.create 17 () in
  let rec f_rec (t : M.t) =
    match Hashtbl.find_opt tbl t with
    | Some res -> res
    | None ->
      let res = f f_rec t in
      Hashtbl.replace tbl t res;
      res
  in
  f_rec

(** Fold over a Memory Tree, up to hash-consing.
    Each labeled function (f_leaf, ...) is applied to the relevant construct.
*)
let fold ~f_bud ~f_leaf ~f_par ~f_switch =
  let f f_rec t =
    let acc = match t with
      | Leaf l -> f_leaf l
      | Bud (ty, l) -> f_bud ty l
      | Par trs -> f_par @@ List.map (fun tr -> f_rec tr) trs
      | Switch { discr; path; cases; default } ->
        f_switch discr path
          (List.map (fun (z, tr) -> z, f_rec tr) cases)
          (Option.map f_rec default)
    in
    acc
  in
  memo_rec f
```

Listing 6: excerpt from Ribbit: hash-consing on Memory Trees.

Memoization On top of hash-consing, and as described in Section 5.6.1, we memoize each call to Seek and Rebuild with the Wrap algorithm. The different functions making up the `Compile_adt` module have a nearly one-to-one correspondence with the various compilation procedures defined in Chapter 5. The actual implementation of Wrap uses a polymorphic hash table to avoid code duplication, but is otherwise identical to the formal algorithm in every point.

Algorithmic checking of types Most of our compilation algorithms are directly translated into our implementation. However, the implementation of formal agreement criteria between high-level and memory types, as well as intrinsic validity and kinding of memory types (all defined in Chapter 3), is not as clear-cut. First, these judments are not defined in a procedural way. Furthermore, especially for agreement, we allow ourselves to quantify over rather unreasonable sets (e.g., every single bit of every primitive for coverage).

The current Ribbit prototype implements a *limited* version of agreement checking. In particular, it verifies agreement with the granularity of a whole memory word and simply accepts any splitting of atoms (as used, notably, in our RISC-V example).

6.4 Visualizing execution traces

The mutually recursive nature of our Seek and Rebuild compilation procedures, presented in Chapter 5, makes it difficult to know precisely which parts of the input expression and memory type are responsible in the event of a crash during this stage of compilation.

To ease debugging of these two algorithms' implementation, Ribbit includes a logger to emit an execution trace in Chrome trace format. Coupled with a trace visualization tool such as Perfetto (`https://ui.perfetto.dev/`), it allows to observe the full stack of nested calls to various compilation functions. Such a trace is shown in Fig. 6.2. It shows the stack of Seek and Rebuild calls, along with various auxiliary functions corresponding to smaller pieces of our compilation algorithms. For each call, we can observe its arguments in the bottom window.

In addition to debugging the implementation per se, this tool also informed some aspects of the actual Rebuild and Seek algorithms, most notably how to anonymize their arguments so that recursive nodes are emitted precisely when necessary.

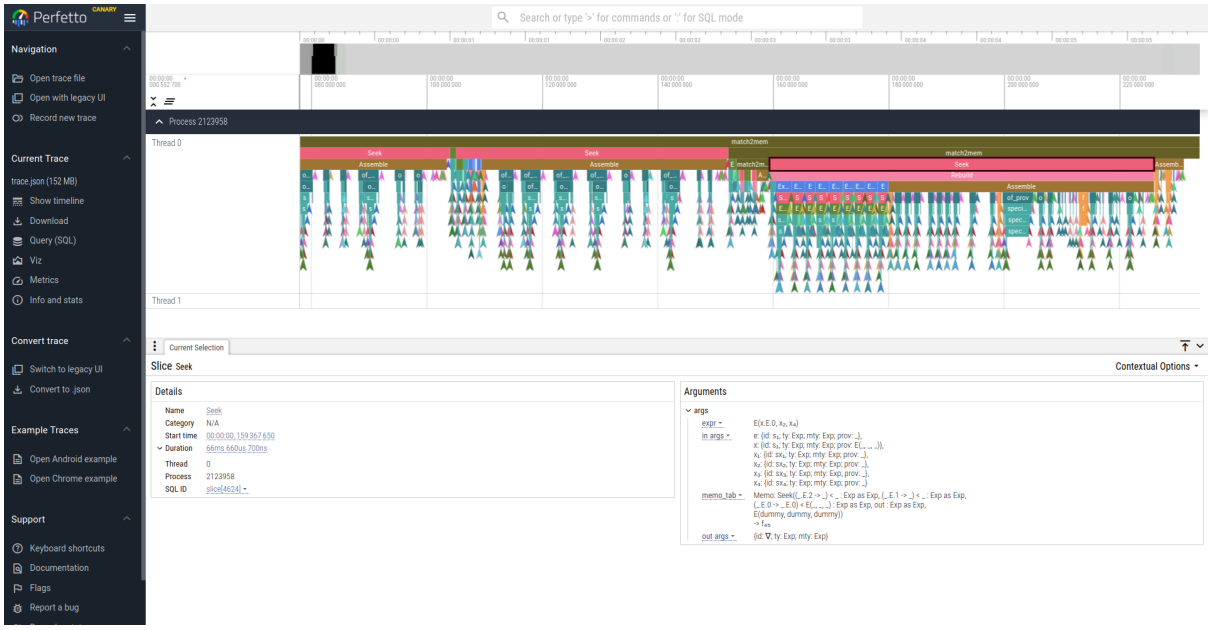


Figure 6.2: Part of a debugging trace for the arithmetic expressions example, in Perfetto.

6.5 In memoriam: the legacy LLVM backend

The formal compilation approach described in Chapters 4 and 5 outputs an expression following the syntax of an abstract target language. Similarly, the current version of the Ribbit implementation compiles input programs this our custom target program representation. It also features an interpreter for this target language. Our target IR is the end of the current compilation chain: the Ribbit prototype lacks a proper backend to either native assembly or a standard low-level intermediate representation. However, at some point during Ribbit development, our implementation did in fact have a working LLVM IR backend. This backend allowed us to run the few benchmarks covered in Chapter 7. In this section, we describe the legacy LLVM IR backend, explain why it was eventually abandoned and explore some possibilities for a future Ribbit backend that would once again allow our implementation to produce executable programs.

The global code generation procedure is rather straightforward: constructors are turned into memory allocation and initialization code and decision trees are turned into LLVM control flow graphs. Two difficulties remain: deal with LLVM's explicitely typed IR and generate code for memory paths.

6.5.1 Values and memory allocation

Words, pointers and struct values correspond to basic low-level memory structures. However, memory values only describe some arrangement of data in memory; where and how we may allocate memory to hold this data remains unspecified. The choice of allocation scheme highly depends on the language and its memory management policy. In Ribbit, all data is allocated on the "main" stack frame, which remains

available during pattern matching execution. The object under scrutiny is then passed by reference to the compiled pattern matching function.

6.5.2 Lowering Memory Types

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

Since every memory value in the decision tree is expressed as a position within the root memory value, we can deduce their types from the root memory type. However, our memory types contain *splits*, which are not expressable as LLVM types (*LLVM Language Reference Manual* 2023). Indeed, LLVM types are unambigous, in that they only describe fully concrete types (in terms of bit width, legal operations, etc.) and do not capture multiple branches. For our purposes, they consist of

1. fixed-width integer types, such as `i32`
2. an opaque pointer type `ptr`
3. structures that aggregate other LLVM types, such as `{i64, ptr}`

. For simplicity, we assume LLVM pointer types are by default 64-bit wide with 8 unused bits due to address alignment. Different pointer types requires using LLVM *address spaces* with different data layout specifications (in the LLVM sense).

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

Lowering split types to LLVM IR types is less immediate. We need an LLVM type that is able to store values of any branch, and in which the split discriminant is accessible. By definition, the exact shape of a memory value instanciated from a split type (and an unknown source value) is unknown until we inspect its discriminant. Type validity ensures this is possible: indeed, a split type has one kind and each branch type must be of this kind. Furthermore, the discriminant location is always accessible in each branch type. Conservatively, we use the "largest" branch type to determine the common type shape. If the kind is **Word**, every branch maps to an LLVM integer type and we take the largest. If the kind is **Block**, every branch maps to an LLVM struct and we recursively find a common type for each field, and keep extra fields. After inspecting the discriminant, we refine the memory type and cast the value to a more precise LLVM type in order to perform operations specific to the identified variant. By validity of memory types, this cast should always be valid.

6.5.3 Code Generation for Memory Paths

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

Example 6.1. The path $.*$ on $\&_{64,8}(\dots)$ becomes

```
%x1 = and i64 %x0, 0xfffffffffffffff00
%x2 = inttoptr i64 %x1 to ptr
%x3 = load ty, ptr %x2
```

△

6.5.4 Possible future backends

The LLVM backend was written at a time when Ribbit was only able to compile the recognition aspect of pattern matching (corresponding to Destruct/to Chapter 4), a very limited subset of variable accessors (corresponding to Extract) and constant value constructors (corresponding to Construct). As such, its output prior to going through the LLVM backend consisted of decision trees (for pattern matching), memory paths (for accessors) and constant memory values (for data constructors).

Since then, we have extended our compilation approach to cover the entire Ribbit language, as described in detail in Chapter 5 (corresponding to Compile, Seek and Rebuild). This required us to define a proper target language (the DPS IR which we defined in Chapter 5) specifically integrated with our formalism, to allow for the complex manipulation of memory types and values which underlie Rebuild and Seek.

Our target IR is richer and more complex than the previous combination of decision trees, memory paths and constant memory values. It also comes with its own memory model. As such, it quickly outgrew the existing LLVM backend, which was subsequently abandoned. Specifically, LLVM IR features its own low-level type system, which restricts the possible data casts in a way which is incompatible with the code emitted by our new compilation algorithms (Refine). In particular, Ribbit's richer type system allow casts that are rejected by LLVM, such as casting an uninitialized 512bit words into a struct.

Nevertheless, it would of course be desirable to restore Ribbit's ability to emit executable code in the future. Doing so would involve writing a backend translating our custom target IR to (either assembly but nobody does that anymore or) a standard IR from a common compiler framework. Possible targets include LLVM IR, but also others such as C−− (S. L. P. Jones, Ramsey, and Reig 1999) or WebAssembly (Wikipedia 2024b). In particular, C−− seems less opinionated than LLVM on possible casts and does not have its own memory type system, thus would probably be easier to interface with Ribbit than LLVM.

Chapter 7

Experimental evaluation of pattern matching compilation

This chapter presents an experimental evaluation of the pattern matching compilation procedure presented in Chapter 4. These experiments were run using a version of the Ribbit prototype which is available as an artifact associated with (Baudon, Radanne, and Gonnord 2023), available at https://doi.org/10.5281/zenodo.7994178 [1]. This version includes the now-defunct LLVM backend, which allowed us to measure concrete execution times.

The experiments we describe here have been performed on a laptop running Gentoo GNU/Linux x86-64 on an Intel Core i5 CPU @ 1.60GHz ×8 (although Ribbit is single-threaded), with 32 GiB of RAM. As for compiler versions, we used Rust 1.67.1, OCaml 4.14.0 and LLVM 14. We demonstrate that:

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

For this purpose, we consider two examples: the red-black trees motivating example from Section 2.1, and a stack machine interpreter example used in (Maranget 2008) to showcase various heuristics for OCaml pattern matching compilation. The full programs are both available in the artifact. For these two examples, we have implemented native OCaml and Rust versions, two Ribbit versions mimicking the internal memory representations of OCaml and Rust, along with the Linux-like red-black tree encoding from Section 2.1.4. However, we do not attempt to match compilation heuristics.

[1] Which was given the "available" as well as "reusable" ACM badges, see https://www.acm.org/publications/policies/artifact-review-and-badging-current

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

Figure 7.1: Execution times on 10^9 runs, and memory usage (exec. time is in ns, memsize in words).

In Fig. 7.1, we first compare the memory usage and number of pointers of some concrete values (a small stack program and a red-black tree with 30 nodes), to cross-validate our memory layout specifications (the size of objects in OCaml/Rust versus the size obtained in Ribbit with our encoding). As we can see, the results are coherent for all implementations. We then compare execution-time performance of the target code generated by each compiler. This comparison should be made while keeping in mind that this version of Ribbit emits code which does far less than native OCaml or Rust: our prototype implementation did not implement sophisticated memory management, or even proper function calls. Additionally, the measured times are very tiny, making measurement difficult. Our goal here is only to show that our pattern matching compilation technique can emit code of similar efficiency to seasoned industry-ready compilers using their memory layouts, which is indeed the case. The final lesson from these dynamic measurements is that improving the memory representation of values is very worthwhile: each reduction from the OCaml representation to the Rust and then to the Linux layout significantly reduces memory footprint and execution times. The Linux red-black trees implementation, in particular, is extremely efficient while having a tiny footprint, demonstrating the gain of such sophisticated bit-stealing.

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

Figure 7.2: Static metrics of decision trees generated by Ribbit : code size is the total number of switches; we also compute the number of switches per path in the decision tree and the number of dereferences along these paths.

Figure 7.2 depicts static metrics obtained via Ribbit for different memory layouts. For both benchmarks, the better performance obtained with the Rust layout and to a further extent with the Linux layout seems to correlate with fewer derefencing operations. The case of red-black trees is even more interesting: despite a greater number of switches and similar path lengths in decision trees, the performance of the Linux encoding still achieves better performance. Perhaps unsurprisingly, it seems that a good static measurement to predict performance is the amount of indirection. These results suggest that we should complement existing heuristics for pattern matching compilation with new ones taking data layout-related metrics into account, such as the number of dereferencing operations and cache-friendliness.

Chapter 8

Related Work

Ribbit is not the only language allowing to describe the memory layout of values. In Section 8.1, we will showcase other approaches to language design which allow programmers to fine-tune, to various degrees, the memory representation of data. We will then present some sources of inspiration. A first source is the rich literature on pattern matching compilation, which was described in Section 4.5. A second inspiration is the variety of memory representations that are used in practice by programmers and language designers. Some of them were already presented in Chapter 2, and we showcase others in Section 8.2.

8.1 Language Approaches to Memory Layout Specification

We first consider language-integrated approaches which allow programmers to design the complete memory representation of some objects. This has been done for numerous purposes, either for performance and control, as is the case for Ribbit, or for verification. It can even arise from the combination of various independent language features.

Low-level Types with High-level Views One approach to achieve precise control over memory layout is to combine two features: an expressive type language allowing to describe the representation, and the ability to create "smart constructors", which hide the low-level representation behind a high-level presentation.

The most complete example of such an approach is probably the Habit language (Diatchki, M. P. Jones, and Leslie 2005), described as "a pure functional language that explores the intersection of low-level programming problems and high-level programming paradigms" [1]. It introduces the notion of *bitdata*: "bit-level representations of data that are required in the construction of many different applications, including operating systems, device drivers, and assemblers." The main idea is to combine two complementary language features: the bitdata memory specification language on the one hand, and *views* (P. Wadler 1987) on the other hand to provide high-level constructors.

As described by Diatchki and M. P. Jones (2006), and unlike Ribbit, it is an extension of Haskell which extends and leverages its rich type system to capture very low-level aspects including alignment and placement in virtual and physical address spaces. Similar to Ribbit's agreement criteria, they describe an informal property dubbed "no junk and confusion", although they do not provide any decision procedure to check that property beyond a small static analysis able to emit warnings in some cases. However, they do not detail their compilation approach ((Hasp) 2013), delegating most delicate compilation problems to an IR called Fidget.

These two features – low-level memory specifications and views – may also be combined in other languages. For instance, active patterns (Syme, Neverov, and Margetson 2007) and pattern synonyms (Pickering et al. 2016) allow users to abstract over patterns by exposing "constructors" which do not directly reflect the underlying definition of the algebraic data type. This allows for both a "programmer" view

[1] http://www.habit-lang.org/

and a "representation" view, similar to our approach. Combined with a rich type algebra with unboxing annotations, it allows some representation tricks. Similarly, (Solodkyy, Reis, and Stroustrup 2013) propose patterns-as-library for C++ based on objects and template meta-programming. Combined with the precise low-level constructs available in C data types, this provides a way to (do the same stuff) in C++.

Verification-oriented approaches Many of the links between ADTs and low-level programming were initially made for *verification*. Notably, Dargent (Chen et al. 2023) allows to specify memory representations in an external DSL which outputs C code for accessors, and Isabelle/HOL theorems; with the aim of formally verifying embedded systems. (Swamy et al. 2022) propose a similar approach to formally verify binary format parsers in F*. Simonnet, Lemerre, and Sighireanu (2023) provide a rich type system capturing a very complete selection of low-level aspects of memory contents, and verify them using abstract interpretation. All these approaches are precise and very expressive, but do not provide language-integrated constructs such as pattern matching. They also also provide far less optimizations than Ribbit.

Memory Layout Optimizations for ADTs Some general-purpose languages with ADTs also provide ways to improve data layout. As we have seen in Section 2.6.4, Rust provides users with a choice between some pre-defined memory representations; in addition, it performs semi-automatic layout optimizations via the notion of *niche* (*RFC: Alignment niches for references types* 2021) to exploit unassigned values. It would be interesting to combine these existing language-level tools with precise data layout annotations such as Ribbit's memory types to provide users with even greater control over memory representation.

OCaml provides a standard language extension featuring an `[@@unboxed]` annotation to *unbox* sum types with a single constructor. Such sum types do not require any extra precaution to unbox: as there is only one possible constructor, there is no need to distinguish between different cases. Even though this standard extension is limited to this very simple situation, Chataing et al. (2024) propose to extend it to a wider variety of scenarios. They propose a sufficient condition and an associated static analysis to ensure that such unboxed constructors do not lead to "confusion" – i.e., using Ribbit terminology, non-distinguishability. This generalization of constructor unboxing still restricts data layout enough to require minimal changes to existing compilation algorithms.

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

Memory layouts for arrays, while largely out of scope of this thesis, have been explored in numerous high-performance languages. For instance, Accelerate [2] (Chakravarty et al. 2011) is an embedded DSL to express idiomatic array transformations in Haskell. At the source level, it allows users to explicitly specify the precise shape of each array. During compilation, it automatically applies memory layout optimizations such as converting "array-of-structs" to "struct-of-arrays". Finally, its CUDA backend must automatically deal with various array shapes to emit efficient GPU code.

Some approaches (Bhaskaracharya, Bondhugula, and Cohen 2016) based on the polyhedral model (Feautrier 1991) allow to automatically optimize the memory layout of array-based data using techniques such as array compaction.

8.2 Optimized Memory Representations

Another important source of inspiration is the plethora of representation tricks and design choices in programming languages to suit various needs. These serves as motivating examples. In an ideal world, Ribbit should be able to express all such representations! In Chapter 2, we showed some specific

[2] https://www.acceleratehs.org/

representation tricks, along with some generic language representation. Here, we showcase some feature-specific designs.

In particular, memory representation in functional polymorphic garbage-collected languages was quickly identified as an important area for performance improvements (Peterson 1989; Leroy 1992; S. L. P. Jones and Launchbury 1991). Our work encourages new development in this area, as it easily supports such representation and allows experimenting with new representations easily.

Unboxing and arrays has been the subject of numerous work and libraries (see (Keller et al. 2010) for a recent Haskell example). We believe many of the data-layout proposed in these works would enhance our approach, notably regarding mutability and concurrency, which we do not explore. Colin, Lepigre, and Scherer (2018) refine the criterion for recursive yet unboxed types in the OCaml case.

Iannetta, Gonnord, and Radanne (2021) and Koparkar et al. (2021) propose completely flattened representations for recursive types, which provide excellent cache behavior and parallelism but require whole-program transformations. In contrast, our technique provides great manual control over memory representation and follow a more traditional compilation pipeline. Supporting such fully flattened layouts in Ribbit would be highly desirable.

Several approaches try to mix polymorphism with optimized data layout. Leroy (1990) shows how to make polymorphic and monomorphic representations work conjointly and Hall, S. L. P. Jones, and Sansom (1994) show how to marry specialization and unboxing. Classically, C++ and Rust rely aggressively on specialization. All these approaches would be compatible with our work.

Chapter 9

Conclusion and future work

This thesis presented the Ribbit language, and demonstrated its usefulness to capture optimized memory layouts for Algebraic Data Types thanks to its fine-grained memory types. The Ribbit compiler lets programmers use high-level, safe language constructs while still reaping the performance benefits of custom memory layouts. Abstractions such as pattern matching are translated to equivalent target code which properly manipulates data following the specified (and possibly quite convoluted) memory representation, thanks to compilation algorithms which follow the structure of memory types.

Let us now consider some possible extensions to our formalism, which would broaden its scope of application and allow it to capture finer, lower-level representations.

Memory management strategies Chapter 5 presented a compilation approach to emit target code which allocates and initializes memory structures to properly represent data. However, it did not explore the topic of deallocation, and our target program representation does not distinguish between deep and shallow copy of memory contents. This last aspect would be particularly important to consider in order to extend Ribbit to work with mutable data.

In the future, we hope to investigate memory management strategies, for instance following Lorenzen, Leijen, and Swierstra (2023) which prevent unnecessary memory allocation, allowing to potentiate the memory usage and performance benefits already associated with optimized memory layouts.

Another possibility would be to leverage stack allocation by encoding the distinction between data allocated on the stack and on the heap within memory types.

Richer memory layouts As shown in Section 2.7, some existing memory layouts (e.g., NaN-boxing) rely on characteristics of non-integer primitive encodings (e.g., using NaN float values in NaN-boxing), which Ribbit does not currently support. More importantly, our current approach is unable to check that memory layouts using techniques such as NaN-boxing or niches are correct (i.e., that such memory types are valid, well-kinded and agree with the associated ADT). Doing so would require extending our formalism to capture architecture- and system-specific details of numeric encodings and machine addresses, which we currently view as completely opaque data.

Another feature which is completely absent from Ribbit is data linearization and array-based layouts. Extending Ribbit with array types, as sketched in (Baudon, Radanne, and Gonnord 2023), would allow users to express "struct-of-arrays" and "array-of-structs" representations, and more generally combine optimized array-based memory layouts with ADTs.

High-level language features So far, Ribbit only supports monomorphic ADTs. However, data types such as finger trees (Hinze and Paterson 2006) (or many of Okasaki's data structures) which require polymorphic recursion do not yield themselves well to monomorphization.

We believe defining memory layouts for polymorphic types is absolutely feasible, following the literature on the topic (Leroy 1990). A simple approach would be the OCaml trick: polymorphic data is always one word (immediate, or pointer), which allow easily emitting polymorphic code. Extending Ribbit with richer high-level types would allow modeling such nested data types.

Automatic synthesis of memory layouts In the future, we also hope to synthesize memory representations automatically, given some specific metrics to optimize. Such metrics might include memory usage or number of pointer dereferences, as mentioned in Chapter 7.

Bibliography

(Hasp), The High Assurance Systems Programming Project (2013). *A Compilation Strategy for the Habit Programming Language*. Tech. rep. URL: http://web.cecs.pdx.edu/~mpj/pubs/compiler.pdf.

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

Aitken, William (1992). "The SML/NJ match compiler". Summer internship report. URL: http://www.smlnj.org/compiler-notes/matchcomp.ps (visited on 06/25/2024).

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

Augustsson, Lennart (1985). "Compiling Pattern Matching". In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. Ed. by Jean-Pierre Jouannaud. Vol. 201. Lecture Notes in Computer Science. Springer, pp. 368–381. DOI: 10.1007/3-540-15975-4_48. URL: https://doi.org/10.1007/3-540-15975-4%5C_48.

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

Baudon, Thaïs, Gabriel Radanne, and Laure Gonnord (June 2022a). "Knit&Frog: Pattern matching compilation for custom memory representations (doctoral session)". In: *AFADL 2022 - 21ème journées Approches Formelles dans l'Assistance au Développement de Logiciels*. Vannes, France. URL: https://hal.inria.fr/hal-03676356.

— (Nov. 2022b). *Ribbit (software)*. URL: https://hal.inria.fr/hal-03860442.

— (Aug. 2023). "Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types". In: *ACM Program. Lang., International Conference on Functional Programming (ICFP)* 7. DOI: 10.1145/3607858. URL: https://doi.org/10.1145/3607858.

— (June 2024). "Compiling Morphisms of Algebraic Data Types". working paper or preprint. URL: https://hal.science/hal-04601882.

Bhaskaracharya, Somashekaracharya G., Uday Bondhugula, and Albert Cohen (Apr. 2016). "Automatic Storage Optimization for Arrays". In: *ACM Trans. Program. Lang. Syst.* 38.3. ISSN: 0164-0925. DOI: 10.1145/2845078. URL: https://doi.org/10.1145/2845078.

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

Cardelli, Luca (1984). "Compiling a functional language". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, pp. 208–217. ISBN: 0897911423. DOI: 10.1145/800055.802037. URL: https://doi.org/10.1145/800055.802037.

Chakravarty, Manuel M T et al. (Jan. 2011). "Accelerating Haskell array codes with multicore GPUs". In: *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM.

Chataing, Nicolas et al. (Jan. 2024). "Unboxed Data Constructors: Or, How cpp Decides a Halting Problem". In: *Proc. ACM Program. Lang.* 8.POPL. DOI: 10.1145/3632893. URL: https://doi.org/10.1145/3632893.

Chen, Zilin et al. (Jan. 2023). "Dargent: A Silver Bullet for Verified Data Layout Refinement". In: *Proc. ACM Program. Lang.* 7.POPL. DOI: 10.1145/3571240. URL: https://doi.org/10.1145/3571240.

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

De Nicola, Rocco and Frits Vaandrager (Mar. 1995). "Three logics for branching bisimulation". In: *J. ACM* 42.2, pp. 458–487. ISSN: 0004-5411. DOI: 10.1145/201019.201032. URL: https://doi.org/10.1145/201019.201032.

Diatchki, Iavor S. and Mark P. Jones (2006). "Strongly typed memory areas programming systems-level data structures in a functional language". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell '06. Portland, Oregon, USA: Association for Computing Machinery, pp. 72–83. ISBN: 1595934898. DOI: 10.1145/1159842.1159851. URL: https://doi.org/10.1145/1159842.1159851.

Diatchki, Iavor S., Mark P. Jones, and Rebekah Leslie (2005). "High-level views on low-level representations". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: Association for Computing Machinery, pp. 168–179. ISBN: 1595930647. DOI: 10.1145/1086365.1086387. URL: https://doi.org/10.1145/1086365.1086387.

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

Feautrier, Paul (1991). "Dataflow analysis of array and scalar references". In: *Int. J. Parallel Program.* 20.1, pp. 23–53. DOI: 10.1007/BF01407931. URL: https://doi.org/10.1007/BF01407931.

Fessant, Fabrice Le 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*. Ed. by Benjamin C. Pierce. ACM, pp. 26–37. DOI: 10.1145/507635.507641. URL: https://doi.org/10.1145/507635.507641.

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

Glabbeek, Rob J. van and W. Peter Weijland (May 1996). "Branching time and abstraction in bisimulation semantics". In: *J. ACM* 43.3, pp. 555–600. ISSN: 0004-5411. DOI: 10.1145/233551.233556. URL: https://doi.org/10.1145/233551.233556.

Hall, Cordelia V., Kevin Hammond, et al. (1992). "The Glasgow Haskell Compiler: A Retrospective". In: *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992*. Ed. by John Launchbury and Patrick M. Sansom. Workshops in Computing. Springer, pp. 62–71. DOI: 10.1007/978-1-4471-3215-8_6. URL: https://doi.org/10.1007/978-1-4471-3215-8%5C_6.

Hall, Cordelia V., Simon L. Peyton Jones, and Patrick M. Sansom (1994). "Unboxing using Specialisation". In: *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, September 12-14, 1994*. Ed. by Kevin Hammond, David N. Turner, and Patrick M. Sansom. Workshops in Computing. Springer, pp. 96–110. DOI: 10.1007/978-1-4471-3573-9_7. URL: https://doi.org/10.1007/978-1-4471-3573-9%5C_7.

Hinze, Ralf and Ross Paterson (2006). "Finger trees: a simple general-purpose data structure". In: *Journal of Functional Programming* 16.2, pp. 197–217. DOI: 10.1017/S0956796805005769.

Iannetta, Paul, 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*. Ed. by Eli Tilevich and Coen De Roover. ACM, pp. 123–129. DOI: 10.1145/3486609.3487204. URL: https://doi.org/10.1145/3486609.3487204.

Jones, Simon L. Peyton 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*. Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, pp. 636–666. DOI: 10.1007/3540543961_30. URL: https://doi.org/10.1007/3540543961%5C_30.

Jones, Simon L. Peyton, Norman Ramsey, and Fermin Reig (1999). "C–: A Portable Assembly Language that Supports Garbage Collection". In: *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*. Ed. by Gopalan Nadathur. Vol. 1702. Lecture Notes in Computer Science. Springer, pp. 1–28. DOI: 10.1007/10704567_1. URL: https://doi.org/10.1007/10704567%5C_1.

Keller, Gabriele et al. (2010). "Regular, shape-polymorphic, parallel arrays in Haskell". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, pp. 261–272. DOI: 10.1145/1863543.1863582. URL: https://doi.org/10.1145/1863543.1863582.

Koparkar, Chaitanya et al. (2021). "Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations". In: *Proc. ACM Program. Lang.* 5.ICFP. DOI: 10.1145/3473596. URL: https://doi.org/10.1145/3473596.

Kosarev, Dmitry, 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*. Ed. by Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, pp. 293–310. DOI: 10.1007/978-3-030-64437-6_15. URL: https://doi.org/10.1007/978-3-030-64437-6%5C_15.

Laville, Alain (May 1991). "Comparison of priority rules in pattern matching and term rewriting". In: *J. Symb. Comput.* 11.4, pp. 321–347. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(08)80109-5. URL: https://doi.org/10.1016/S0747-7171(08)80109-5.

Leroy, Xavier (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*. Ed. by Pierre Deransart and Jan Maluszynski. Vol. 456. Lecture Notes in Computer Science. Springer, pp. 255–276. DOI: 10.1007/BFb0024189. URL: https://doi.org/10.1007/BFb0024189.

— (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*. Ed. by Ravi Sethi. ACM Press, pp. 177–188. DOI: 10.1145/143165.143205. URL: https://doi.org/10.1145/143165.143205.

[SW] Leroy, Xavier and Antoine Miné, *Zarith* 2010. LIC: LGPL-2.0-only WITH OCaml-LGPL-linking-exception. SWHID: ⟨swh:1:dir:56fa3bba77ee6c8a40b4230eba576600641a4db1;origin=https://github.com/ocaml/Zarith;visit=swh:1:snp:3f57262eeea9a011da7b93770318fc6411728b6a;anchor=swh:1:rev:e67bccbe69362866d2ce182bb4ba8a1a458cd387⟩.

Liu, Fengyun (2016). "A generic algorithm for checking exhaustivity of pattern matching (short paper)". In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA 2016. Amsterdam, Netherlands: Association for Computing Machinery, pp. 61–64. ISBN: 9781450346481. DOI: 10.1145/2998392.2998401. URL: https://doi.org/10.1145/2998392.2998401.

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

Lorenzen, Anton, Daan Leijen, and Wouter Swierstra (Aug. 2023). "FP²: Fully in-Place Functional Programming". In: *Proc. ACM Program. Lang.* 7.ICFP. DOI: 10.1145/3607840. URL: https://doi.org/10.1145/3607840.

MacQueen, David (2022). "A New Match Compiler for Standard ML of New Jersey". In: *ML workshop 2022*. ACM. URL: https://icfp22.sigplan.org/details/mlfamilyworkshop-2022-papers/3/A-New-Match-Compiler-for-Standard-ML-of-New-Jersey (visited on 06/25/2024).

Maranget, Luc (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*. Ed. by Jon L. White. ACM, pp. 21–31. DOI: 10.1145/141471.141499. URL: https://doi.org/10.1145/141471.141499.

— (1994). "Two Techniques for Compiling Lazy Pattern Matching". In: URL: https://api.semanticscholar.org/CorpusID:61076221.

— (2007). "Warnings for pattern matching". In: *J. Funct. Program.* 17.3, pp. 387–421. DOI: 10.1017/S0956796807006223. URL: https://doi.org/10.1017/S0956796807006223.

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

Minsky, Yaron and Anil Madhavapeddy (2021). "Real World OCaml". In: chap. Memory Representation of Values. URL: https://dev.realworldocaml.org/runtime-memory-layout.html.

Peterson, John (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*. Ed. by Joseph E. Stoy. ACM, pp. 89–99. DOI: 10.1145/99370.99377. URL: https://doi.org/10.1145/99370.99377.

Pettersson, Mikael (1992). "A Term Pattern-Match Compiler Inspired by Finite Automata Theory". In: *Proceedings of the 4th International Conference on Compiler Construction*. CC '92. Berlin, Heidelberg: Springer-Verlag, pp. 258–270. ISBN: 3540559841.

Pickering, Matthew et al. (2016). "Pattern synonyms". In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. by Geoffrey Mainland. ACM, pp. 80–91. DOI: 10.1145/2976002.2976013. URL: https://doi.org/10.1145/2976002.2976013.

Puel, Laurence and Ascánder Suárez (1993). "Compiling Pattern Matching by Term Decomposition". In: *J. Symb. Comput.* 15.1, pp. 1–26. DOI: 10.1006/jsco.1993.1001. URL: https://doi.org/10.1006/jsco.1993.1001.

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

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

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

Sekar, R. C., R. Ramesh, and I. V. Ramakrishnan (Dec. 1995). "Adaptive Pattern Matching". In: *SIAM J. Comput.* 24.6, pp. 1207–1234. ISSN: 0097-5397. DOI: 10.1137/S0097539793246252. URL: https://doi.org/10.1137/S0097539793246252.

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

Shaikhha, Amir et al. (2017). "Destination-Passing Style for Efficient Memory Management". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: Association for Computing Machinery, pp. 12–23. ISBN: 9781450351812. DOI: 10.1145/3122948.3122949. URL: https://doi.org/10.1145/3122948.3122949.

Simonnet, Julien, Matthieu Lemerre, and Mihaela Sighireanu (July 2023). "A dependent nominal physical type system for the static analysis of memory in low level code". working paper or preprint. URL: https://hal.science/hal-04649674.

Solodkyy, Yuriy, 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*. Ed. by Jaakko Järvi and Christian Kästner. ACM, pp. 33–42. DOI: 10.1145/2517208.2517222. URL: https://doi.org/10.1145/2517208.2517222.

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

Syme, Don, 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*. Ed. by Ralf Hinze and Norman Ramsey. ACM, pp. 29–40. DOI: 10.1145/1291151.1291159. URL: https://doi.org/10.1145/1291151.1291159.

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

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

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

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

Vollmer, Michael et al. (2019). "LoCal: a language for programs operating on serialized data". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, pp. 48–62. DOI: 10.1145/3314221.3314631. URL: https://doi.org/10.1145/3314221.3314631.

Wadler, P. (1987). "Views: a way for pattern matching to cohabit with data abstraction". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, pp. 307–313. ISBN: 0897912152. DOI: 10.1145/41625.41653. URL: https://doi.org/10.1145/41625.41653.

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

Waterman, Andrew et al. (Dec. 2019). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 20191213*. Tech. rep. RISCV fundation.

[SW exc.], *WebKit NaN-boxing* 2023. SWHID: ⟨swh:1:cnt:1eec01b377e2c5e9f6a1b9ba27bdc322919256dd;origin=https://github.com/WebKit/WebKit;visit=swh:1:snp:df63564a304482d8c3aa219d0c9a1ce47dee6573;anchor=swh:1:rev:e7780d735de3975ec8bf854e4ed70d07c7648497;path=/Source/JavaScriptCore/runtime/JSCJSValue.h;lines=402⟩.

Wikipedia (2024a). *Red–black tree — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&oldid=1237143863. [Online; accessed 29-July-2024].

— (2024b). *WebAssembly — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=WebAssembly&oldid=1231319980. [Online; accessed 30-July-2024].