Algebraic Data Type and Nested Pattern Matching (L6)

Owen Li ∩ Yicheng Zhang
Friendship Ave Team

May 2023

1 Introduction

In this project, we propose the implementation of Algebraic Data Types (ADT), variant constructions, and nested pattern-matching. Our project allows users to define custom `enum` types and create instances of specific variants of these `enum` types. Users can match an expression of type `enum T` onto a nested pattern, and destructure the expression based on the pattern. We also introduce a novel approach to exhaustive match checking of nested types: the *Duo-Colored Lazy Matching Tree* (DLMT), which employs lazy computation to verify whether nested types are matched exhaustively.

Implementation-wise, at least from a high-level, an expression e of type `enum T` is a pointer that points to a chunk of memory M, which holds an integer specifying its corresponding variant, as well as a chunk of raw memory M'. Construction of a specific variant, as in statement `x = T::Variant(e1, ..., en)`, is performed via allocating a chunk of memory M, writing its corresponding variant tag to the start of M, evaluating each of `e1, .., en`, and writing the results one by one to the remaining parts of the M. Pattern-matching of some e of type `enum T` is performed via fetching the aforementioned variant tag, checking if it matches the variant as specified by the given pattern, and then destructuring the variant by moving suitable parts of M into a list of temporaries. There are a lot more implementation details, to be further discussed in Section 3.

The results have been satisfying, as it makes C0 much easier to use: each variant of some `enum T` can capture a list of data of any "small-types", including `enum T` itself, which enables easy implementations of data-structures like lists and trees; the construction of `enum` variants automatically acquires heap memory, so users do not have to worry about memory while programming; the support for nested patterns also make usage more intuitive. Overall, these powerful constructs give C0 a somewhat "functional" taste, and makes modeling complex datatypes / datastructures much more ergonomic, as compared to what we had to suffer through in some imperative programming class like CMU 15-122. (Sorry, Iliano!)

2 Specification

We first give a syntax specification of L6, then separately discuss syntax and rules for declaring `enum` types, construction instances of `enum`'s, and pattern-matching.

2.1 Syntax

Below is our L6 syntax in Backus Naur Form. The blue parts are what we added on top of L4. Un-modified parts are abbreviated.

⟨program⟩ ::= ϵ | ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩ ::= ⟨fdecl⟩ | ⟨fdef⟩ | ⟨typedef⟩ | ⟨sdecl⟩ | ⟨sdef⟩ | ⟨edef⟩
⟨edef⟩ ::= **enum ident =** ⟨variant-list⟩ **;**
⟨variant⟩ ::= **ident** ⟨typlist⟩
⟨variant-list-follow⟩ ::= ϵ | **|** ⟨variant⟩ ⟨variant-list-follow⟩
⟨variant-list⟩ ::= ϵ | ⟨variant⟩ ⟨variant-list-follow⟩
⟨typlist-follow⟩ ::= ϵ | **,** ⟨typ⟩ ⟨typlist-follow⟩
⟨typlist⟩ ::= ϵ | **(** ⟨typ⟩ ⟨typlist-follow⟩ **)**
⟨type⟩ ::= **int** | **bool** | **ident** | ⟨type⟩ ***** | ⟨type⟩ **[]** | **struct ident** | **enum ident**
⟨control⟩ ::= **if (** ⟨exp⟩ **)** ⟨stmt⟩ ⟨elseopt⟩ | **while (** ⟨exp⟩ **)** ⟨stmt⟩
| **for (** ⟨simpopt⟩ **;** ⟨exp⟩ **;** ⟨simpopt⟩ **)** ⟨stmt⟩
| **return** ⟨exp⟩ **;** | **return ;**
| **assert (** ⟨exp⟩ **)** | **match** ⟨exp⟩ **{** ⟨arms⟩ **}**
⟨arms⟩ ::= ϵ | ⟨arm⟩ ⟨arms⟩
⟨arm⟩ ::= ⟨patt⟩ `=>` ⟨block⟩
⟨patt⟩ ::= **ident :: ident** ⟨patt-list⟩ | **ident** | _
⟨patt-list-follow⟩ ::= ϵ | **,** ⟨patt⟩ ⟨patt-list-follow⟩
⟨patt-list⟩ ::= ϵ | **(** ⟨patt⟩ ⟨patt-list-follow⟩ **)**
⟨exp⟩ ::= **(** ⟨exp⟩ **)** | **num** | **true** | **false** | **ident** | **NULL** | ⟨unop⟩ ⟨exp⟩
| ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ | **ident** ⟨arg-list⟩
| ⟨exp⟩ **. ident** | ⟨exp⟩ `->` **ident** | **alloc (** ⟨exp⟩ **)** | ***** ⟨exp⟩
| **alloc (** ⟨type⟩ **,** ⟨exp⟩ **)** | ⟨exp⟩ [⟨exp⟩]
| **ident :: ident** | **ident :: ident (** ⟨arglist⟩ **)**

2.2 Enum Declaration

To declare an enum type `enum T`, we first declare the type and then associate to it a list of `variant`s, separated by vertical lines, like the following syntax:

```
enum Tree = Node(enum Tree, int, enum Tree) | Leaf;
```

Each variant of `enum` can optionally capture a list of data, whose types are specified between the pair of parenthesis.

Note that only small types are supported, so letting a variant capture a struct, like the following,

```
enum Maybe = Exist(struct Foo) | Null; // not ok
```

is not allowed.

Note `enum XXX` is considered a first-class member, and is therefore supported in C0 structs, pointers, arrays, etc:

```
struct Forest {
    enum Tree[] trees;
    enum Tree* favorite_tree;
    int num_trees;
}
```

2.3 Enum Construction

To construct something of an enum type, we must specify both its type and its variant, like the following:

```
enum Tree t1 = Tree::Node(Tree::Leaf, 0, Tree::Leaf);
```

The constructor can take arbitrary expression of correct type as its arguments.

In the above example, construction happened 3 times: first two constructions took place while evaluating the `Tree::Leaf` expressions, and once all three arguments of the `Tree::Node` variant are evaluated, the data of `Tree::Node` variant is finally constructed.

2.4 Pattern Matching Syntax

Pattern-matching provides a powerful form of control-flow. The syntax for pattern-matching looks like the following:

```
int sum_tree(enum Tree t) {
    match t {
        Tree::Leaf => { return 0; }
        Tree::Node(t1, n, t2) => { return sum_tree(t1) + n + sum_tree(t2); }
    }
}
```

In particular, a `match` statement consists of an expression to be match onto (in this case, `t`), and a list of "arms" (in this case, `Tree::Leaf => {...}` and `Tree::Node(...) => {...}`.) Each arm consists of a pattern to match onto, and a body of statements to execute.

In the above example, once matched onto the second branch, variables `t1` and `t2` will be automatically declared and initialized, so the match-arm body can directly use these variables.

2.5 Pattern and Nested Pattern

A pattern can be one of the following forms:

1. A single variable,
2. A wildcard `_`,
3. A specific variant `Typ::Variant` without arguments, and
4. A specific variant with arguments, i.e. `Typ::Variant(p1, p2, ..., pn)`, where each `pi` is itself a pattern.

This means that nested patterns are allowed, so the following

```
Tree::Node(Tree::Node(_, _, _), _, _)
```

is a valid pattern.

2.6 Pattern Matching Order

Note that pattern matching searches for a valid pattern starting from the top, so the following

```
match t {
    _ => { return 0; }
    _ => { return 1; }
}
```

will return `0`.

2.7 Exhaustiveness Checking

`match` statements with non-exhaustive branches will not pass the type-checker. For example, the following

```
enum Bool = Yes | No;
int bool_to_num(enum Bool b) {
    match b {
        Bool::Yes => { return 1; }
    }
}
```

will not pass the type-checker, because the `Bool::No` branch is missing.

3 Implementation

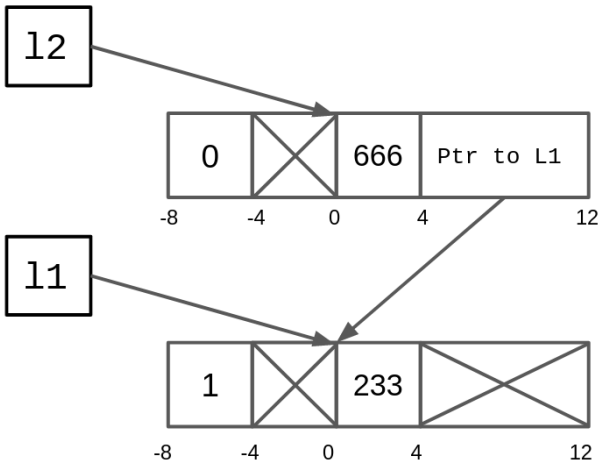
We will separately discuss the memory layout of `enum`, as well as the implementation details of constructor, pattern-matching, and destructuring.

3.1 Memory layout of `enum`

Our implementation of `enum` is essentially a pointer that points to a chunk of memory M, consisting of two segments: (1) a tag; and (2) a chunk of raw memory, which we call "data segment". The tag tells the program which `variant` of the `enum` shall the chunk of memory in (2) be interpreted as. For example, if we define the following,

```
enum List = Body(int, enum List) | End(int);
enum List l1 = End(233);
enum List l2 = Body(666, l1);
```

then the memory layout looks like the following:



Each of `l1`, `l2` takes up 20 bytes of memory on heap. The 4 bytes in $[-8, -4)$ holds an `int`, which we call a "tag", that tells the specific variant this memory corresponds to, where 0 means the first variant of the `enum`, and 1 means the second variant of the `enum`, etc. The next 4 bytes in $[-4, 0)$ are for alignment only, and thus crossed off in the diagram.

(We let the pointers point to the middle of the memory chunk, while storing the tags within $[-8, -4)$, because this is exactly how array-lengths in C0 are stored. Using the same scheme allows L4 safemode array bounds-check and L6 fetching enum tag to share the same code)

From here on, things get a bit interesting: the 12 bytes within $[0, 12)$ could represent different things, depending on the "tag". For example, in the case of `l2`, these 12 bytes should be used to represent the list of data captured by variant `List::Body` (first variant, corresponds to tag 0), namely an `int`, and an `enum List`, i.e. a pointer. Therefore, the first 4 bytes in $[0, 4)$ correspond to the `int` and holds the value "666", while the next 8 bytes in $[4, 12)$ holds a pointer that points

to the memory of `l1`. In the case of `l1`, though, these 12 bytes only need to be used to represent the list of data captured by variant `List::End` (second variant, corresponds to tag 1), namely an `int`, so the first 4 bytes in $[0, 4)$ holds "233", while the next 8 bytes are simply unused.

3.2 Constructor

When a constructor of an `enum` variant is used, as in the following example,

```
enum List l1 = Body(233);
```

the program performs the following steps under the hood:

1. Allocates a chunk M of heap memory of sufficient size (i.e. the largest size any of the variants will ever need), where pointer p points to the beginning of M.

2. Writes the corresponding tag of the variant (in this case 1) to the first four bytes of M

3. If the variant constructor has a list of expressions $e_1, ..., e_n$ as its arguments (in this case, "233"), then for each e_i from left to right, evaluate e_i and write the result to the corresponding memory location in M. Specifically, the evaluations of each e_i is packed together, arranged corresponding to their order in C0 source code, and flushed to the left, immediately after the first 8 bytes of M.

4. Increment p by 8 (ie. let it point to the start of data-segment) and return p as the "output" of the constructor.

3.3 Pattern Matching and Destructuring

Given an expression `e` of some `enum` type, we can perform pattern-matching on it. Note that each pattern is a possibly nested complex of variants, variables, and wildcards. Given some `match e {...}`, our compiler generates assembly code that tries to "unify" `e` with the pattern in each arm, jumps to the arm body upon success, and jumps to the next unification upon failure.

For unification of expression `e`, stored in some `temp_0`, the assembly performs an algorithm that looks like the following pseudo-code:

```
unify(p, temp_0):
    # Since p is known at compile time, exactly one branch
    # of this `if-else` will actually turn into assembly code.
    if p = Variant(p1, p2, ..., pn):
        temp_e <- deref(temp_0 - 8)     # fetch tag of e
        temp_p <- tag of Variant        # known at compile time

        if temp_e == temp_p:
            # unification of top-level pattern succeeds. Need to destructure e,
            # i.e. move stuffs from its memory segment into a bunch of temps,
            # and then perform unifications with the subpatterns p1, ..., pn.

            # these offsets are known at compile time.
            temp_1 <- deref(temp_0 + offset_1)
            ......
            temp_n <- deref(temp_0 + offset_n)

            # recursively unify the destructured expression
            if not unify(p1, temp_1):
                return false
            ......
            if not unify(pn, temp_n):
                return false

            # unification of all sub-patterns succeed.
            return true
        else:
            # unification fails.
            return false
    else if p = Variable(x):
        # if p is a variable, we simply create a new temp for x in codegen...
        temp_x <- temp_0        # ...initialize it with temp_0, the evaluation of e
        return true             # ...and unification succeeds.
    else:     # p = wildcard
        return true
```

Basically, it checks if `e`, represented as a pointer stored in `temp_0`, has the correct tag; then it destructures the memory segment of `temp_0` into a list of temps, recursively tries to unify them with the sub-patterns of `p`, and initializes the variables along the way.

If we closely examine the above pseudocode, we notice that much of the info like nested-depth of pattern, enum memory offsets, etc. are simply known at compile time, therefore it is not hard to convert the above pseudo-code into assembly code. In fact, you can find our implementation in the file `codegen_elab2.rs`, within the function `unify()`.

As for correctness, suppose a match arm `p => {body}` performs unification `unify(p, e)`. On success, all variables declared in `p` are correctly initialized via destructuring `e`; if unification fails halfway through, then since C0 disallows variable shadowing, any variable modified during unification will have a scope exactly equal to `body`, which is never executed because unification fails. This means a failed unification will not cause unexpected side-effects.

3.4 Match Exhaustiveness Checking

3.4.1 Complexity analysis

Note that Exhaustiveness Checking is NP-hard, as we can reduce 3SAT to it via Karp-Reduction: for a 3SAT problem with n variables and m clauses, we define the following enum:

```
enum Bool = T | F;
enum Tuple = Tup(enum Bool, enum Bool, ..., enum Bool)
```

i.e. the `enum Tuple` is some n-tuple of booleans. We use $p_{i,j,k}$ to denote some pattern `Tup(...)` where the i, j, kth position are `Bool::F` and everything else is a wildcard. Then we create a `match` statement, and for each clause $(x_i \vee x_j \vee x_k)$, we add an arm to the `match` statement with pattern $p_{i,j,k}$. If and only if the 3SAT has a valid boolean assignment $b_1, ..., b_n$, there will be no 3-clause with all its variables false, and therefore the pattern `Tup(b_1, ..., b_n)` unifies with none of the match arms, i.e. non-exhaustive.

3.4.2 Duo-Colored Lazy Matching Tree (DLMT)

We introduce a novel *Duo-Colored Lazy Matching Tree* (DLMT), which features two types of nodes: an *Unmatched Variant* (`UnMatched`) and a *Tuples* node. Both node types can be lazy, meaning they will only expand their children during the matching process.

3.4.3 `Tuples`

A `Tuple` represents one element in the set of a matching variant tuples. It may have descendants, which are its peers at the same level. The `Tuple` can operate in two modes: expanded or lazy. In lazy mode, it does not contain any unmatched variants; instead, it functions as a node within a tree. While in this mode, it may have descendants as they might need to be matched with other peers in the tuple set (see the leftmost tree in Figure 1). Upon expansion, the node's children become its variants, and its descendants are moved to become the variants' descendants. Each child now has a separate chain of descendants, necessitating a deepcopy.

When variants are successfully matched, the corresponding descendants can be deleted, and the node can be marked as matched, eliminating the need to traverse the chain again.

3.4.4 `UnMatched`

An `UnMatched` node serves as a variant of the `Tuple` and can operate in either lazy or expanded mode. In lazy mode, it possesses no descendants other than those inherited from its parent `Tuples`. During expansion, new `Tuples` are added as descendants to the `UnMatched` node.

Once all of its descendant tree nodes are matched, the `UnMatched` node can be collapsed back into lazy mode.

3.4.5 Adding

During the addition process, we first expand the `Tuples` we are working with and then attempt to match each variant. When matching a variant results in multiple entries within the tuple set, we traverse down the tree to complete the matching process. After fitting all elements into

their appropriate positions, we assess whether the nodes are matched and, if so, shrink the tree accordingly.

3.4.6 Checking Exhaustiveness After Addition

Given that the tree expands and shrinks during the addition process, we can determine the exhaustiveness by checking if the root node is in a matched state.

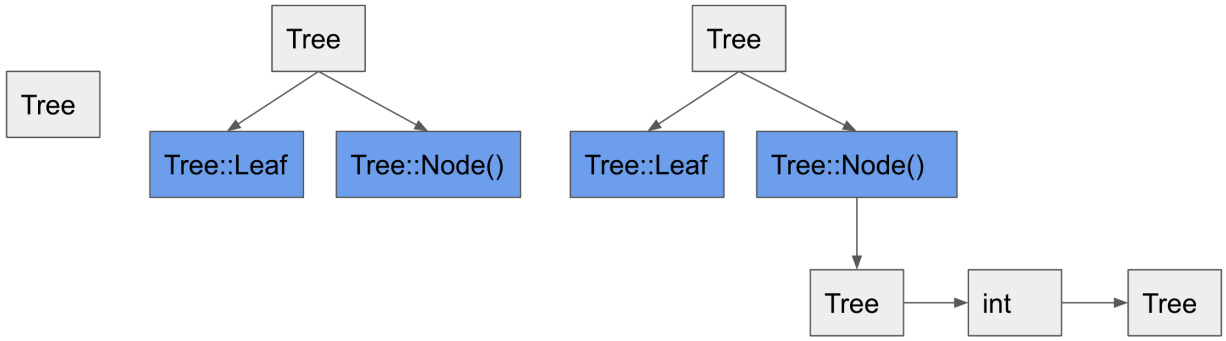


Figure 1: Example check of `Tree`: Lazy Tree Root, unexpanded tree variant nodes, and lazy tree node tuples

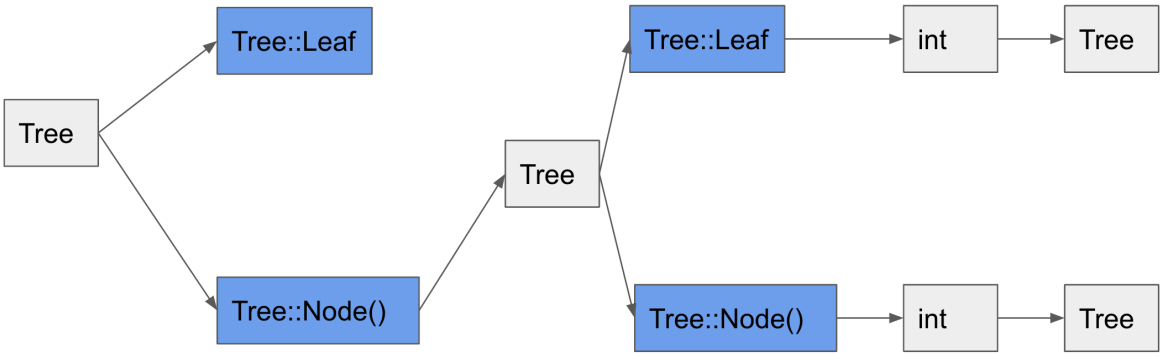


Figure 2: Example check of `Tree`: Partial expanded tree node tuples

4 Testing

Since our implementation is mainly feature-focused, our testing emphasized the correctness of `enum`'s behaviors under a wide range of cases. Specifically, we need to ensure that (1) our new features work well with existing C0 features; (2) each of our deliberately-designed features behave as intended; and (3) our `enum` + `match` features behave correctly when used to implement real-world data structures and algorithms. We designed the tests that fall into any of the following categories:

1. Tests that simulates how algebraic data types (`enum`'s) would be used in a real-world scenario, like representing tree / linked-list data structures.

2. Tests that checks whether the newly-added data type works well with pre-existing C0 types, like putting an `enum` in a `struct`, or capturing a pointer inside an `enum`.

3. Tests that contains deeply-nested pattern matches, which ensures that nested pattern unification is robust.

4. Tests that contains somewhat nontrivial `match` cases with possibly missing arms, which ensures that match exhaustiveness check behave as intended.

5. Tests that contains possibly malformed `enum` constructors and match patterns, which ensures that the type checker correctly catches misuses of the `enum` / `match` feature.

As for testing infrastructure, we basically re-used the scheme for previous labs. Specifically, we annotated each of our test-cases with its intended output, and then directly invoke the `./gradecompiler` executable on our test case directory. Our test is in `./l6-tests/`. To run the tests, simply go to the `/compiler` directory and run the following command:

```
make test
```

If you use OS X, you will need to output slightly different assembly (error codes and labels will be different). Please go to `./lab6/src/util/c0spec.rs` and change `MAC` to be true.

Since our testing approach covers both hypothetical and real-world uses of the ADT feature, each individual functionality, and how these functionalities work with existing C0 features, we can be fairly confident that if all tests passes, then our implementation functions as expected.

5 Analysis

Overall, this compiler is a success: it not only produces correct (i.e. passed all correctness test cases) and reasonably performant (i.e. got most of the points in L5) assembly code for C0, but also supports additional features like algebraic data type, nested pattern matching, exhaustiveness checking. Its user-ergonomics is also to be noted, as the frontend parser can literally draw an arrow that points to where things went wrong, like below:

```
Expected SEMICOLON, got PLUS:
9 |   int x + ;
            ^
```

This compiler is not without drawbacks, though, and some future improvements can be made. For one, the type-checker, the code generation pass, and the asm-to-x86 convertion passes are not as modular as they could have been, and we believe that refactoring these parts will make the compiler more maintainable. Next, the efficiency of some compiler algorithms, like liveness propagation and type-checking, can also be improved via eliminating some of the `xxx.clone()`'s (a natural consequence of implementing our compiler in Rust^{TM}). In addition, our compiler does not implement some key optimizations such as the SSA form or dataflow analysis, so the output assembly is still noticeably slower as compared to that of `gcc -O1`. Implementing these parts will make our compiler competitive in terms of the performance of generated assembly code. Furthermore, if a user writes `C0` source code that can be successfully parsed as an AST but fails type-checking, the type-checker would have a hard time pinpointing the specific line of source code that goes wrong (understandable, since it only has access to an elaborated AST). We believe that implementing some type of map between source code and elaborated AST can help the type-checker print out useful hints if typechecking fails. Finally, while our pattern matching currently supports the key features of nested variants, variables, and wildcards, but it does not yet support additional modern features like matching onto integer / boolean literals, or matching into an array. These features shall be reasonably easy to implement, but would certainly make `C0` a more flexible and user-friendly language.