Compiling pattern matching

Yorick Peterse
https://yorickpeterse.com

About me

- Using Rust since 2015 👴
- Working on a programming language: https://inko-lang.org
  - Rust is used for the native code compiler and a small runtime library
- Before that: GitLab, and a few other small companies
  - Most memorable achievement: solving GitLab's scaling problems by removing the production database, then finding out we didn't have any backups
- See https://yorickpeterse.com for more information

Inko

- Statically typed, compiled to machine code using LLVM
- Single ownership and move semantics, but no lifetimes so it’s easier to use compared to Rust
- Mixes elements from Rust, Erlang, and Pony
- Competes with the likes of Erlang, Go, Java, etc
- Supports pattern matching

Inko: Hello World

```
import std.stdio.STDOUT

class async Main {
  fn async main {
    STDOUT.new.print("Hello, world!")
  }
}
```

Inko: doubly-linked lists

```
class Node[T] {
  let @value: T
  let @next: Option[Node[T]]
  let @prev: Option[mut Node[T]]
}

class List[T] {
  let @head: Option[Node[T]]
  let @tail: Option[mut Node[T]]
}
```

Pattern matching in Inko

```
let value = Option.Some(("foo", 42, Person { @name = "Alice" }))

match value {
  case Some(("foo", num, { @name = "Alice" })) -> { ... }
  case Some(("bar", 50, _)) -> { ... }
  case _ if some_condition -> { ... }
  case _ -> { ... }
}
```

Implementing pattern matching

- Existing implementations are difficult to understand
- Books are either horribly outdated, or don’t cover the subject properly
- Papers on the subject are difficult to read
- Some resources only cover code generation, not exhaustiveness checking

Implementing pattern matching

- “How to compile pattern matching” by Jules Jacobs, 2021
- Inko implements this algorithm
- Gleam is also implementing this algorithm: https://github.com/gleam-lang/gleam/pull/2091
- https://github.com/yorickpeterse/pattern-matching-in-rust implements this algorithm in simple Rust, along with another algorithm

Terminology

- **Pattern**: what we want to match, e.g. Some((42, "test"))
- **Variable**: a "thing" to test against, typically generated by the compiler
- **Binding**: a name to assign a matching value to
- **Column**: a pattern to test, and a variable to test it against
- **Row**: columns, an optional guard, and the code to execute upon a match
- **Constructor**: something that creates an instance of a finite type ("true", "Some", tuples, etc)
- **Variant**: the possible values/cases of an enum, e.g. Some and None

Algorithm overview

1. The match expression is turned into a list of rows and columns
    a. Not covered
2. Bindings are pushed out of rows and columns, into the code to run
3. Each column has an explicit variable to test against
4. The algorithm turns these into an efficient "decision tree"
5. The decision tree is verified for exhaustiveness
6. The tree is turned into executable code
    a. Not covered

Decision tree

- It's as boring as it sounds: a tree where nodes are decisions, and edges outcomes 🤯
- In the context of pattern matching, nodes are tests to perform and edges the results of the tests
- Nodes:
  - **Success**: pattern matched, run some code
  - **Failure**: pattern is missing, i.e. the match isn't exhaustive
  - **Guard**: checks a guard, branches based on the result
  - **Switch**: test a variable against one or more patterns, branch accordingly

Decision trees in Rust

```
enum Decision {
  Failure,
  Success(Body),
  Guard(Condition, Body, Box<Decision>)
  Switch(Variable, Vec<Case>, Option<Box<Decision>>)
}
struct Case {
  constructor: Constructor,
  arguments: Vec<Variable>,
  body: Decision,
}
```

Decision trees in Rust

```rust
struct Body {
  bindings: Vec<(String, Variable)>,
  value: usize,
}
```

Patterns in Rust

```
enum Pattern {
  Constructor(Constructor, Vec<Pattern>),
  Int(i64),
  Binding(String),
}
enum Constructor {
  True,
  False,
  Int(i64),
  Variant(TypeId, usize), // usize is the variant index
}
```

Rows and columns

```
struct Row {
  columns: Vec<Column>,
  guard: Option<usize>,
  body: Body,
}

struct Column {
  variable: Variable,
  pattern: Pattern,
}
```

```
struct Variable {
  id: usize,
  type_id: TypeId,
}

struct TypeId(usize);
```

Example patterns

`true`

```
Pattern::Constructor(Constructor::True, Vec::new())
```

`None`

```
Pattern::Constructor(Constructor::Variant(TypeId(42), 0), Vec::new())
```

`Some(true)`

```
Pattern::Constructor(
  Constructor::Variant(TypeId(42), 1),
  vec![Pattern::Constructor(Constructor::True, Vec::new())]
)
```

Example: matching a boolean

```
match x { true => A, false => B }

let t = Pattern::Constructor(Constructor::True, Vec::new());
let f = Pattern::Constructor(Constructor::False, Vec::new());

vec![
  Row::new(vec![Column::new(x, t)], None, A), // None = no guard
  Row::new(vec![Column::new(x, f)], None, B),
]
```

Example: matching a boolean

```
match x { true => A, false => B }

Decision::Switch(
  x,
  vec![
    Case::new(Constructor::False, Vec::new(), B),
    Case::new(Constructor::True,  Vec::new(), A),
  ],
  None
)
```

Where do we begin?

```rust
fn compile(rows: Vec<Row>) -> Decision
  todo!("...")
```

Algorithm overview

1. **Bindings are pushed out of rows and columns, into the code to run**
2. The algorithm turns these into an efficient decision tree
3. The decision tree is verified for exhaustiveness

Pushing bindings out of patterns

- Simplifies the algorithm in a few places
- Columns containing bindings are then removed

Pushing bindings out of patterns

```
for row in &mut rows
  row.columns.retain |col|
    if let Pattern::Binding(bind) = &col.pattern
      row.body.bindings.push((bind.clone(), col.variable));
      false
    else
      true
```

Pushing bindings out of patterns

```
fn compile(rows: Vec<Row>) -> Decision
  for row in &mut rows
    row.columns.retain |col|
      if let Pattern::Binding(bind) = &col.pattern
        row.body.bindings.push((bind.clone(), col.variable));
        false
      else
        true
```

Algorithm overview

1. ~~Bindings are pushed out of rows and columns, into the code to run~~
2. **The algorithm turns these into an efficient decision tree**
3. The decision tree is verified for exhaustiveness

Efficient decision trees

- Don't test the same pattern multiple times
- Avoid duplicating code as much as possible
- We achieve this by using a heuristic to decide what to branch on

```
enum Example {
  Zero,
  Add(i64, i64),
  Mul(i64, i64),
}
```

Branching heuristic

```
match thing {
   Add(Add(x, y), Zero)        => A,
   Add(Mul(x, y), Zero)        => B,
   Add(x,         Mul(y, z))   => C,
   Add(x,         Add(y, z))   => D,
   Add(x,         Zero)        => E,
}
```

For the first pattern, do we branch on the inner `Add(x, y)` (column 1) or `Zero` (column 2)?

Branching heuristic: branch on column 1

```
Add(x, y), Zero        => A
Add(x, y), Mul(y, z) => C
Add(x, y), Add(y, z) => D
Mul(x, y), Zero        => B
Mul(x, y), Mul(y, z) => C
Mul(x, y), Add(y, z) => D
_,         Mul(y, z) => C
_,         Add(y, z) => D
_,         Zero        => E
```

Lines in **red** are tests with duplicated code.

Branching heuristic: branch on column 2

```
Add(x, y), Zero      => A
Mul(x, y), Zero      => B
_,         Zero      => E
_,         Mul(y, z) => C
_,         Add(x, y) => D
```

Lines in **red** are tests with duplicated code.

Branching heuristic: branch on column 1

```
V1 is Add(x, y), V2 is Zero      => A
V1 is Add(x, y), V2 is Mul(y, z) => C
V1 is Add(x, y), V2 is Add(y, z) => D
V1 is Mul(x, y), V2 is Zero      => B
V1 is Mul(x, y), V2 is Mul(y, z) => C
V1 is Mul(x, y), V2 is Add(y, z) => D
_,               V2 is Mul(y, z) => C
_,               V2 is Add(y, z) => D
_,               V2 is Zero      => E
```

Branching heuristic algorithm

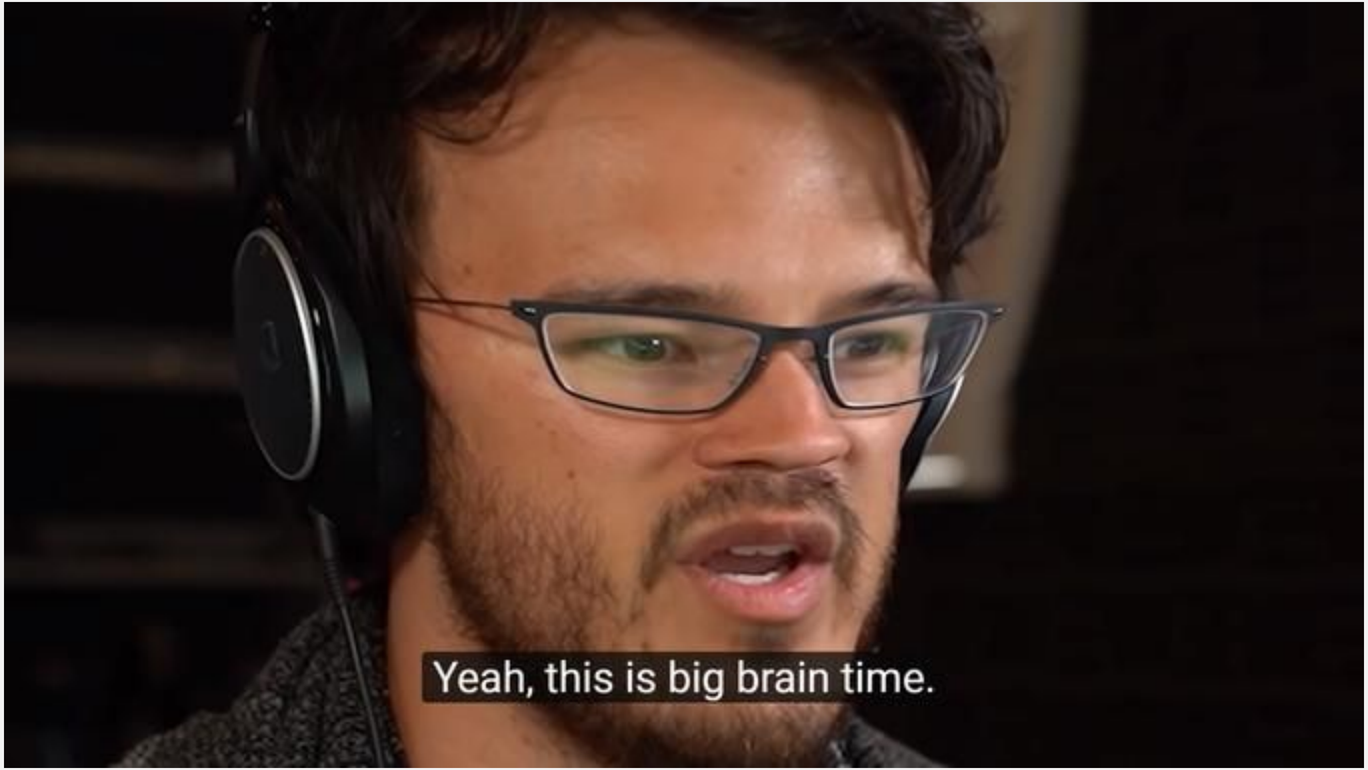
- Count the amount of occurrences of each variable, grouped per variable
- For the first row, branch on the variable with the highest number of occurrences
- Not perfect, but good enough for 95% of the time

Branching heuristic in Rust

```
let mut counts = HashMap::new();

for row in &rows
  for col in &row.columns
    *counts.entry(&col.variable).or_insert(0) += 1;

rows[0]
  .columns
  .iter()
  .map(|c| c.variable)
  .max_by_key(|v| counts[v])
  .unwrap()
```

Compiling the branch

1. Take the column we want to branch on in our row
2. Get the type of the variable, so we know what "strategy" to use (integer literals, constructors, etc)
3. The strategy is a function that takes as input a list of rows, a branching variable, and additional data (e.g. the available constructors)
4. The strategy returns the decision tree
5. Repeat this recursively until we have one remaining row without any columns


Yeah, this is big brain time.

Compiling branches/strategies

```
fn compile(rows: Vec<Row>) -> Decision
  ...

  let branch = branch_variable(&rows);

  match type_of(branch)
    Boolean => ...
    Enum(variants) => ...
    ...
```

Compiling constructors

```
fn constructors(
  rows: Vec<Row>,
  branch: Variable,
  mut cases: Vec<(Constructor, Vec<Variable>, Vec<Row>)>,
) -> Vec<Case>
  ...
```

Compiling constructors: booleans

```
let cases = vec![
  (Constructor::False, Vec::new(), Vec::new()),
  (Constructor::True,  Vec::new(), Vec::new())
];

Decision::Switch(branch, constructors(rows, branch, cases), None);
```

Compiling constructors: enums

```
let cases = Vec::new();

for (idx, types) in variants.iter().enumerate()
  let cons = Constructor::Variant(branch.type_id, idx);
  let vars = create_variables(types);

  cases.push((cons, vars, Vec::new()));

Decision::Switch(branch, constructors(rows, branch, cases), None);
```

Compiling constructors: implementation

```
for mut row in rows
  if let Some(col) = row.remove_column(&branch)
    ...
  else
    for (_, _, rows) in &mut cases
      rows.push(row.clone());

cases
  .into_iter()
  .map(|(cons, vars, rows)| Case::new(cons, vars, compile(rows)))
  .collect()
```

Compiling constructors: implementation

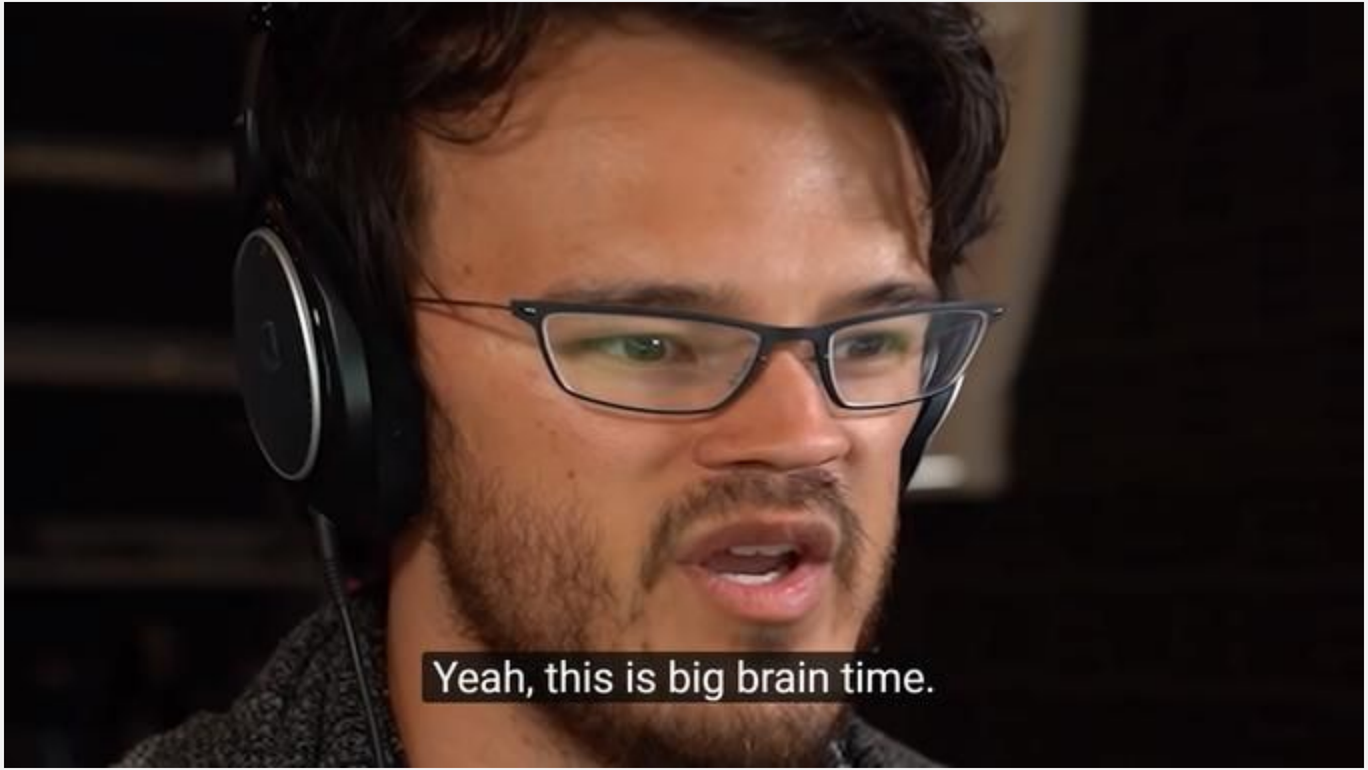
```
// cases: Vec<(Constructor, Vec<Variable>, Vec<Row>)>

if let Pattern::Constructor(cons, args) = col.pattern
  let idx      = cons.index();
  let mut cols = Vec::new();

  for (var, pat) in cases[idx].1.iter().zip(args)          A
    cols.push(Column::new(var, pat));

  cases[idx].2.push(Row::new(cols, row.guard, row.body));   B
```

Terminating the branch

```
fn compile(rows: Vec<Row>) -> Decision
  ... move bindings code ...

  if rows.first().map_or(false, |c| c.columns.is_empty())
    let row = rows.remove(0);

    if let Some(guard) = row.guard
      return Decision::Guard(guard, row.body, Box::new(compile(rows)));
    else
      return Decision::Success(row.body);

  ... strategy code ...
```

Algorithm overview

1. ~~Bindings are pushed out of rows and columns, into the code to run~~
2. ~~Each column has an explicit variable to test against~~
3. ~~The algorithm turns these into an efficient decision tree~~
4. **The decision tree is verified for exhaustiveness**


Yeah, this is big brain time.

Verifying exhaustiveness

```
fn compile(rows: Vec<Row>) -> Decision
  if rows.is_empty()
    exhaustive = false;
    return Decision::Failure;

  ... rest of the code discussed thus far ...
```

Result

```
fn compile(rows: Vec<Row>) -> Decision
  if rows.is_empty()
    ... Check exhaustiveness ...
  move_bindings(rows);

  if rows.first().map_or(false, |c| c.columns.is_empty())
    ... Code that checks if we're done ...

  let branch = branch_variable(&rows);

  match type_of(branch)
    ... Handle the different strategies ...
```

Result

```
let rows        = lower_input_to_rows(input);
let exhaustive = true;
let tree        = compile(rows);

if exhaustive
  compile_to_executable_code(tree);
else
  error();
```

What remains?

- Additional patterns: OR, ranges, tuples, integers, etc
- Compiling to executable code
- Checking for redundant patterns
- Generating a list of missing patterns for an error message
- https://github.com/yorickpeterse/pattern-matching-in-rust covers all that, minus code generation, scan the QR code for easy access

Questions?