First Committee Meeting

Progress Report

Jason Balaci

McMaster University

Oct. 21st, 2021

Table of Contents

1. Introduction

2. Project
   - Drasil
   - Goal #1: Typed Expression Language
   - Goal #2: Model Discrimination – "ModelKinds"

3. References

Table of Contents

1. Introduction

2. Project
   - Drasil
   - Goal #1: Typed Expression Language
   - Goal #2: Model Discrimination – "ModelKinds"

3. References

Who am I?

- Jason Balaci

Who am I?

- Jason Balaci
- Graduate of *McMaster University*, holding...
  - Honours Actuarial and Financial Mathematics (B.Sc.)
  - Minor in Computer Science

Who am I?

- Jason Balaci
- Graduate of *McMaster University*, holding...
  - Honours Actuarial and Financial Mathematics (B.Sc.)
  - Minor in Computer Science
- Currently pursuing a Master's of Computer Science (M.Sc) at *McMaster University*, under the supervision of **Dr. Carette**

Overview of Progression towards Master's of Computer Science (M.Sc.)

Table of Contents

1 Introduction

2 Project
- Drasil
- Goal #1: Typed Expression Language
- Goal #2: Model Discrimination – "ModelKinds"

3 References

Table of Contents

1. Introduction

2. Project
   - Drasil
   - Goal #1: Typed Expression Language
   - Goal #2: Model Discrimination – "ModelKinds"

3. References

What are Disjoint Union Types?

- **Disjoint union types** (DUTs) are types where values can take the form of one of many type constructions, with a **unique identifier tag**.

What are Disjoint Union Types?

- **Disjoint union types** (DUTs) are types where values can take the form of one of many type constructions, with a **unique identifier tag**.
- Each DUT has a set of possible types it can take on, called "variants", or "kinds".

What are Disjoint Union Types?

- **Disjoint union types** (DUTs) are types where values can take the form of one of many type constructions, with a **unique identifier tag**.
- Each DUT has a set of possible types it can take on, called "variants", or "kinds".
- These "variants" may be records, or the unit type ("()").

What are Disjoint Union Types?

- **Disjoint union types** (DUTs) are types where values can take the form of one of many type constructions, with a **unique identifier tag**.
- Each DUT has a set of possible types it can take on, called "variants", or "kinds".
- These "variants" may be records, or the unit type ("()").
- An instance of a DUT may take on the form of **only one** of it's variants.

What are Disjoint Union Types?

- **Disjoint union types** (DUTs) are types where values can take the form of one of many type constructions, with a **unique identifier tag**.
- Each DUT has a set of possible types it can take on, called "variants", or "kinds".
- These "variants" may be records, or the unit type ("()").
- An instance of a DUT may take on the form of **only one** of it's variants.
- They are often found in functional programming languages, where they are usually known as **Algebraic Data Types** (ADTs).

What do they look like?

- Declarations in Haskell,
  ```
  data List a = Nil | Cons a (List a)
  data RGB = Red | Green | Blue
  ```

What do they look like?

- Declarations in Haskell,
```
data List a = Nil | Cons a (List a)
data RGB = Red | Green | Blue
```

- Declarations in our implementation,
```
type List = Nil | Cons(value: integer, tail: List)
type RGB = Red | Green | Blue
```

What do they look like?

- Declarations in Haskell,
```
data List a = Nil | Cons a (List a)
data RGB = Red | Green | Blue
```

- Declarations in our implementation,
```
type List = Nil | Cons(value: integer, tail: List)
type RGB = Red | Green | Blue
```

- Instantiation in Haskell,
```
a = Cons 1 (Cons 2 (Cons 3 Nil))
b = Red
```

What do they look like?

- Declarations in Haskell,
```
data List a = Nil | Cons a (List a)
data RGB = Red | Green | Blue
```
- Declarations in our implementation,
```
type List = Nil | Cons(value: integer, tail: List)
type RGB = Red | Green | Blue
```
- Instantiation in Haskell,
```
a = Cons 1 (Cons 2 (Cons 3 Nil))
b = Red
```
- Instantiation in our implementation,
```
a := Cons(1, Cons(2, Cons(3, Nil())))
b := Red()
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.

```
case <variable> of {
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.

```
case <variable> of {
    [nil:   <stmtSuite>]
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.

```
case <variable> of {
    [nil:  <stmtSuite>]
    Kind A: <stmtSuite>
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.
- Within the statement suite of each variant case, the variable in question is assumed to be an instance of the variant's record.

```
case <variable> of {
   [nil:  <stmtSuite>]
   Kind A: <stmtSuite>
   Kind B: <stmtSuite>
   ...
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.
- Within the statement suite of each variant case, the variable in question is assumed to be an instance of the variant's record.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.

```
case <variable> of {
   [nil:  <stmtSuite>]
   Kind A: <stmtSuite>
   Kind B: <stmtSuite>
   ...
   [default:  <stmtSuite>]
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.
- Within the statement suite of each variant case, the variable in question is assumed to be an instance of the variant's record.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.

```
case <variable> of {
   [nil:  <stmtSuite>]
   Kind A: <stmtSuite>
   Kind B: <stmtSuite>
   ...
   [default:  <stmtSuite>]
   ...  or ...
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.
- Within the statement suite of each variant case, the variable in question is assumed to be an instance of the variant's record.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.

```
case <variable> of {
   [nil:  <stmtSuite>]
   Kind A: <stmtSuite>
   Kind B: <stmtSuite>
   ...
   [default:  <stmtSuite>]
   ...  or ...
   [default nothing]
}
```

How do we use DUTs?

The anatomy of the case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using an optional `nil` case at the start.
- case on any of the variants.
- Within the statement suite of each variant case, the variable in question is assumed to be an instance of the variant's record.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.

```
case <variable> of {
   [nil:  <stmtSuite>]
   Kind A: <stmtSuite>
   Kind B: <stmtSuite>
   ...
   [default:  <stmtSuite>]
   ...  or ...
   [default nothing]
}
```

Cover your cases!

If you create a non-exhaustive case statement, the compiler will warn you.

References

- Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages." *Journal of Functional Programming* 19.5 (2009): 509.