Disjoint Union Types in P0

Project 9 / Group 8

Jason Balaci

McMaster University

April 2021

Table of Contents

Table of Contents

What are Disjoint Union Types?

What do they look like?

Grammar Changes

- Text visible on slide 1

Grammar Changes

- Text visible on slide 1
- Text visible on slide 2

Grammar Changes

- Text visible on slide 1
- Text visible on slide 2
- Text visible on slide 3

Grammar Changes

- Text visible on slide 1
- Text visible on slide 2
- Text visible on slide 3
- Text visible on slide 4

How do we use DUTs?

The anatomy of a case statement.

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

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

How do we use DUTs?

The anatomy of a case statement.

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

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

How do we use DUTs?

The anatomy of a case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using a `nil` case at the start.
- case on any of the variants you'd like to.

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

How do we use DUTs?

The anatomy of a case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using a `nil` case at the start.
- case on any of the variants you'd like to.
- `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>
    ...
    Kind Z: <stmtSuite>
    [default: <stmtSuite>]
    ... or ...
    [default nothing]
}
```

How do we use DUTs?

The anatomy of a case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using a `nil` case at the start.
- case on any of the variants you'd like to.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.
- 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>
    ...
    Kind Z: <stmtSuite>
    [default: <stmtSuite>]
    ... or ...
    [default nothing]
}
```

How do we use DUTs?

The anatomy of a case statement.

- cases are the only way to access data inside of DUTs.
- Check if DUTs were *initialized* using a `nil` case at the start.
- case on any of the variants you'd like to.
- `default` case allows you to perform either a statement suite or a no-op on all non-covered cases.
- 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>
    ...
    Kind Z: <stmtSuite>
    [default: <stmtSuite>]
    ... or ...
    [default nothing]
}
```

Exhaust your cases!

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

Table of Contents

Example: Maybe

```
type Maybe = Just(value: integer)
           | Nothing

procedure valOr(v: Maybe, n: integer) → (r: integer)
    case v of {
        Just:
            r := v.value
        default:
            r := n
    }

program Main
    var maybe: Maybe

    maybe <- Nothing()
    writeln(valOr(maybe, -1))

    maybe <- Just(1111)
    writeln(valOr(maybe, 0))
```

Example: Maybe

```
type Maybe = Just(value: integer)
           | Nothing

procedure valOr(v: Maybe, n: integer) → (r: integer)
    case v of {
        Just:
            r := v.value
        default:
            r := n
    }

program Main
    var maybe: Maybe

    maybe <- Nothing()
    writeln(valOr(maybe, -1))

    maybe <- Just(1111)
    writeln(valOr(maybe, 0))
```

Output

```
-1
1111
```

Example: Lists

```
type List = Cons(head: integer, tail: List)
          | Nil

procedure upToList(n: integer) → (l: List)
    if n < 1 then l := Nil() else l := Cons(n, upToList(n−1))

procedure consumeList(l: List)
    case l of {
        Cons: writeln(l.head); consumeList(l.tail)
        default nothing
    }

procedure sumList(l: List) → (n: integer)
    case l of {
        Cons: n := sumList(l.tail) + l.head
        default: n := 0
    }

program Main
    var myList: List
    myList := upToList(5)
    consumeList(myList)
    writeln(sumList(myList))
```

Example: Lists

```
type List = Cons(head: integer, tail: List)
          | Nil

procedure upToList(n: integer) → (l: List)
    if n < 1 then l := Nil() else l := Cons(n, upToList(n−1))

procedure consumeList(l: List)
    case l of {
        Cons: writeln(l.head); consumeList(l.tail)
        default nothing
    }

procedure sumList(l: List) → (n: integer)
    case l of {
        Cons: n := sumList(l.tail) + l.head
        default: n := 0
    }

program Main
    var myList: List
    myList := upToList(5)
    consumeList(myList)
    writeln(sumList(myList))
```

Output

```
5
4
3
2
1
15
```

Example: Strings

... lists in disguise?

```
type String = SCons(ch: integer, tail: String)
            | SNil

procedure printStr(s: String, ln: boolean)
    case s of {
        SCons: writeChar(s.ch); printStr(s.tail, ln)
        default: if ln then writeNewLine()
    }

// inclusively generating alphabets in a range
procedure genBetwn(start: integer, end: integer) -> (s: String)
    var ch: integer
    ch := end
    s := SNil()

    while start <= end do
        s, start, ch := SCons(ch, s), start + 1, ch - 1

program Main
    // print capital letters
    printStr(genBetwn('A', 'Z'), true)

    // print lowercase letters
    printStr(genBetwn('a', 'z'), true)

    // print numbers 0–9
    printStr(genBetwn('0', '9'), true)

    // print Greek letters
    printStr(genBetwn('α', 'ω'), true)
```

Note

We convert single-quoted characters into UTF-8 integer representation when reading in P0 programs.

Example: Strings

... lists in disguise?

```
type String = SCons(ch: integer, tail: String)
            | SNil

procedure printStr(s: String, ln: boolean)
    case s of {
        SCons: writeChar(s.ch); printStr(s.tail, ln)
        default: if ln then writeNewLine()
    }

// inclusively generating alphabets in a range
procedure genBetwn(start: integer, end: integer) -> (s: String)
    var ch: integer
    ch := end
    s := SNil()

    while start <= end do
        s, start, ch := SCons(ch, s), start + 1, ch - 1

program Main
    // print capital letters
    printStr(genBetwn('A', 'Z'), true)

    // print lowercase letters
    printStr(genBetwn('a', 'z'), true)

    // print numbers 0-9
    printStr(genBetwn('0', '9'), true)

    // print Greek letters
    printStr(genBetwn('α', 'ω'), true)
```

Output

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
$\alpha\beta\gamma\delta\epsilon\zeta\eta\theta\iota\kappa\lambda\mu\nu\xi o\pi\rho\sigma\tau\upsilon\phi\chi\psi\omega$

Note

We convert single-quoted characters into UTF-8 integer representation when reading in P0 programs.

Other Examples

```
type RainbowColour = Red | Orange | Yellow | Green | Blue | Indigo | Violet

type Either = Left(value: integer)
            | Right(value: boolean)

type Tree = Branch(left: Tree, right: Tree)
          | Leaf(value: integer)

type Expr = Add(left: Expr, right: Expr)
          | Sub(left: Expr, right: Expr)
          | Mul(left: Expr, right: Expr)
          | Div(num: Expr, den: Expr)
          | Pow(base: Expr, exponent: Expr)
          | Int(value: integer)

type StringIntMap = SIMCons(key: String, value: integer, tail: StringIntMap)
                  | SIMEmpty
```

Remark

Modelling is nice with disjoint union types!

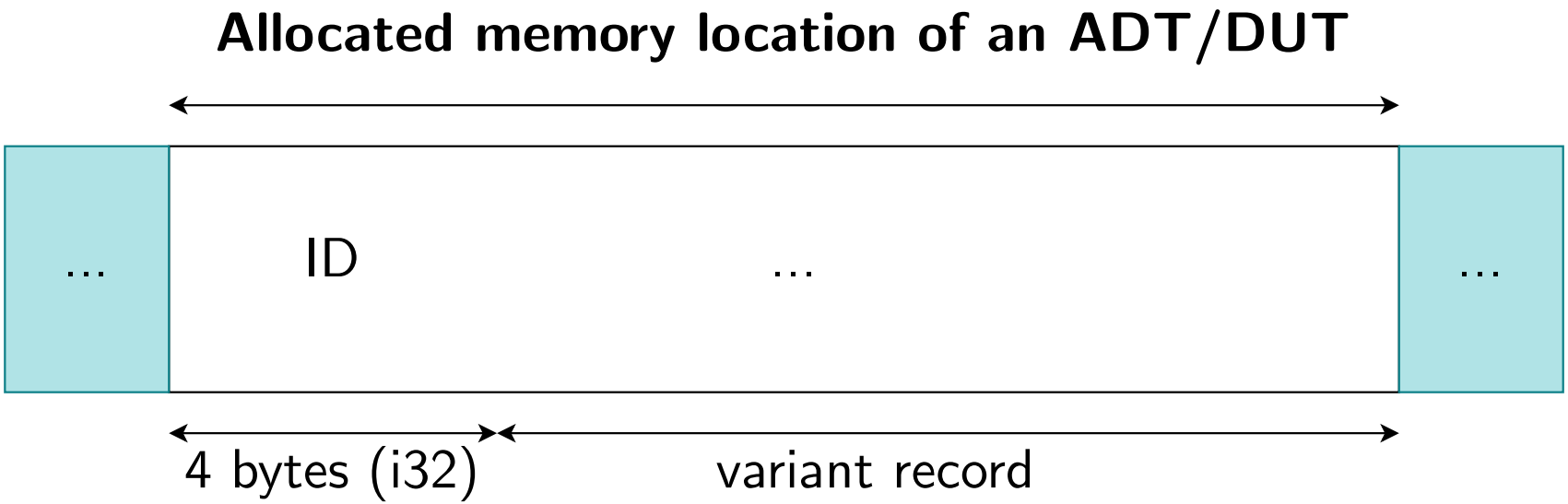
Table of Contents

case WebAssembly Generation

Memory Impact & Management

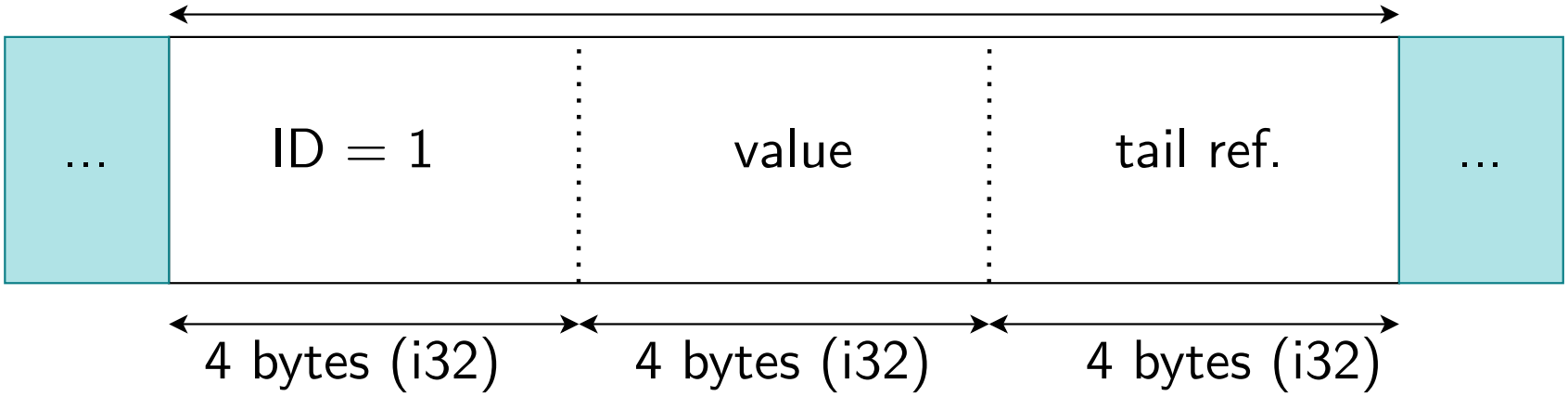
Each instance of a DUT is located on the heap, and instances of local/global DUTs are pointers to the locations of their corresponding DUT on the heap.

- Size of an allocation depends on the size of the variant being instantiated
- Offsets to accessing variables work similar to records, with a 4 byte offset for the variant id.

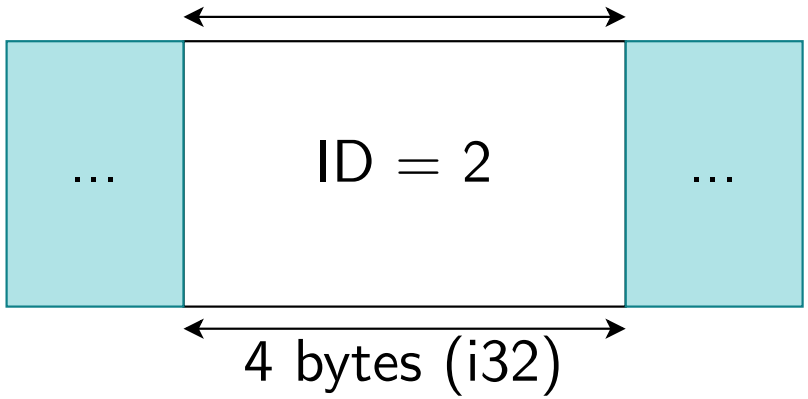


Example: Lists in Memory

Allocated memory location of a ‘Cons‘ (12 bytes)



Allocated memory location of a ‘Nil‘ (4 bytes)



Notable Design Decisions

- The first 4 bytes of a program are to be always initialized to 0 so that we can always have uninitialized DUT pointers pointing to it, then when this uninitialized DUT is read in, we will always see that the "instance" has $kindId = 0$, meaning it's uninitialized (and also not instantiated).

Notable Design Decisions

- The first 4 bytes of a program are to be always initialized to 0 so that we can always have uninitialized DUT pointers pointing to it, then when this uninitialized DUT is read in, we will always see that the "instance" has $kindId = 0$, meaning it's uninitialized (and also not instantiated).
- When DUTs are read in, we create "helper functions" for each DUT variant. These "helper functions" are then used whenever we want to instantiate any particular DUT variant. DUT variant instantiation is hence secretly just a function call (to some function that creates a DUT on the heap and returns it's memory location).

Notable Design Decisions

- The first 4 bytes of a program are to be always initialized to 0 so that we can always have uninitialized DUT pointers pointing to it, then when this uninitialized DUT is read in, we will always see that the "instance" has $kindId = 0$, meaning it's uninitialized (and also not instantiated).
- When DUTs are read in, we create "helper functions" for each DUT variant. These "helper functions" are then used whenever we want to instantiate any particular DUT variant. DUT variant instantiation is hence secretly just a function call (to some function that creates a DUT on the heap and returns it's memory location).
- Disjoint union type variants are mutable records. You may modify the values of a DUT variant only when caseing on it from within it's case.

Notable Design Decisions

- The first 4 bytes of a program are to be always initialized to 0 so that we can always have uninitialized DUT pointers pointing to it, then when this uninitialized DUT is read in, we will always see that the "instance" has $kindId = 0$, meaning it's uninitialized (and also not instantiated).
- When DUTs are read in, we create "helper functions" for each DUT variant. These "helper functions" are then used whenever we want to instantiate any particular DUT variant. DUT variant instantiation is hence secretly just a function call (to some function that creates a DUT on the heap and returns it's memory location).
- Disjoint union type variants are mutable records. You may modify the values of a DUT variant only when caseing on it from within it's case.
- DUT variant kind identifiers are immutable!

Example of DUT instantiation helper

This function is used when wanting to instantiate a “Cons” variant (of a List).

Notes on Runtimes

- When working with DUTs/ADTs, we often tend to create recursive algorithms...

Notes on Runtimes

- When working with DUTs/ADTs, we often tend to create recursive algorithms...
- This causes issues for **pywasm**
  - Due to being interpreted in Python, a recursive call stack size limitation is imposed onto our programs.

Notes on Runtimes

- When working with DUTs/ADTs, we often tend to create recursive algorithms...
- This causes issues for **pywasm**
  - Due to being interpreted in Python, a recursive call stack size limitation is imposed onto our programs.
- Thankfully, **wasmer** has no issues!

Notes on Runtimes

- When working with DUTs/ADTs, we often tend to create recursive algorithms...
- This causes issues for **pywasm**
  - Due to being interpreted in Python, a recursive call stack size limitation is imposed onto our programs.
- Thankfully, **wasmer** has no issues!
- In-browser WebAssembly execution also has no issues, but we don't ship a web browser with the compiler.

Table of Contents

- Type variables!
  - Polymorphic disjoint union types! No more StringLists, IntLists, BooleanLists, etc!
  - More code reuse!

- Type variables!
  - Polymorphic disjoint union types! No more StringLists, IntLists, BooleanLists, etc!
  - More code reuse!
- More built-in types and syntactic sugars
  - Strings, Lists, Maps as a basic set of built-in DUTs
  - Stronger syntactic sugar for String generation (e.g., “abcd...” for quickly instantiating large strings)

- Type variables!
  - Polymorphic disjoint union types! No more StringLists, IntLists, BooleanLists, etc!
  - More code reuse!
- More built-in types and syntactic sugars
  - Strings, Lists, Maps as a basic set of built-in DUTs
  - Stronger syntactic sugar for String generation (e.g., “abcd...” for quickly instantiating large strings)
- Improved Memory Management
  - Memory freeing!
  - Memory reuse!
  - Allocation specialization for built-in DUTs!

References