

# Disjoint Union Types in P0

Project 9 / Group 8

Jason Balaci

McMaster University

April 2021

# Table of Contents

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation
- 4 Future Work
- 5 Conclusion

# Table of Contents

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation
- 4 Future Work
- 5 Conclusion

# 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.

# Table of Contents

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation
- 4 Future Work
- 5 Conclusion

# 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: Integer 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 printList(l: List)
  case l of {
    Cons: writeln(l.head); printList(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)
  printList(myList)
  writeln(sumList(myList))
```



# Example: Integer 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 printList(l: List)
  case l of {
    Cons: writeln(l.head); printList(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)
  printList(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 their 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
αβγδεζηθικλμνξοπρστυφχψω
```

## Note

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

# Other Examples

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

type Maybe = Just(value: integer)
           | Nothing

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
```

# Other Examples

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

type Maybe = Just(value: integer)
           | Nothing

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

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation**
- 4 Future Work
- 5 Conclusion

# Focal Grammar Changes

- Disjoint union type declarations

```
type ::=  
  ident ["(" typedIds ")"] {"|" ident ["(" typedIds ")"]}  
  | ...
```

# Focal Grammar Changes

- Disjoint union type declarations

```
type ::=  
  ident ["(" typedIds ")"] {"|" ident ["(" typedIds ")"]}  
  | ...
```

- case statements

```
statement ::= ... | "case" ident "of" "{" INDENT  
  ["nil" ":" statementSuite]  
  {ident ":" statementSuite}  
  ["default" (":" statementSuite | "nothing")]  
  DEDENT "}"
```



# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., “`m := 1 + pow(2, 3)`”)

# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., “`m := 1 + pow(2, 3)`”)
- Single characters wrapped in single-quotes (e.g., ‘`a`’) is a syntactic sugar for converting single utf-8 characters into P0 integers

# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., “`m := 1 + pow(2, 3)`”)
- Single characters wrapped in single-quotes (e.g., ‘`a`’) is a syntactic sugar for converting single utf-8 characters into P0 integers
- “`< -`” and “`- >`” as alternatives for “`←`” and “`→`”, respectively

# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., “`m := 1 + pow(2, 3)`”)
- Single characters wrapped in single-quotes (e.g., ‘`a`’) is a syntactic sugar for converting single utf-8 characters into P0 integers
- “`< -`” and “`- >`” as alternatives for “`←`” and “`→`”, respectively
- “`>=`” and “`<=`” as alternatives for “`≥`” and “`≤`”, respectively

# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., `m := 1 + pow(2, 3)`)
- Single characters wrapped in single-quotes (e.g., `'a'`) is a syntactic sugar for converting single utf-8 characters into P0 integers
- “< -” and “- >” as alternatives for “←” and “→”, respectively
- “>=” and “<=” as alternatives for “≥” and “≤”, respectively
- “\*” as an alternative for “×”

# Supplementary Grammar and Procedure Changes

- Single-value returning procedures may be used “in-place” in expressions (e.g., `m := 1 + pow(2, 3)`)
- Single characters wrapped in single-quotes (e.g., `'a'`) is a syntactic sugar for converting single utf-8 characters into P0 integers
- `< -` and `- >` as alternatives for `←` and `→`, respectively
- `>=` and `<=` as alternatives for `≥` and `≤`, respectively
- `*` as an alternative for `×`
- Standard procedures
  - `write` - no longer prints a newline character
  - `writeln` - writes single integer to std. out. with a newline afterwards
  - `writeChar` - writes single integer converted into a utf-8 character to std. out.
  - `writeCharLn` - writes single integer converted into a utf-8 character to std. out. with a newline afterwards
  - `writeNewLine` - writes a newline character to std. out.

# Example: case WebAssembly Generation

The WebAssembly code on the right-hand side is generated for the below case statement.

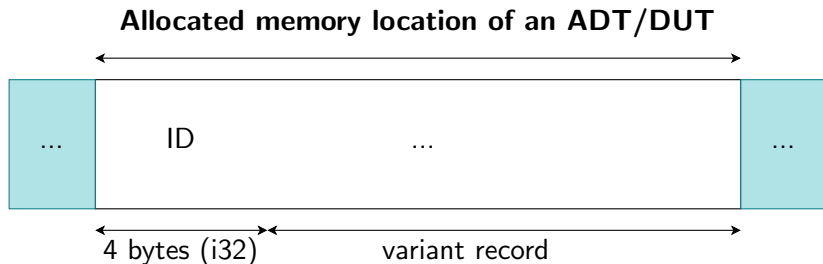
```
type Colour = R | G | Unknown

procedure printCol(col: Colour)
  case col of {
    nil: writeCharLn('?')
    R: writeCharLn('R')
    G: writeCharLn('G')
    default: writeCharLn('?')
  }
```

```
...
local.get $col
i32.load
i32.const 0                                ;; check if nil/0
i32.eq                                     ;; if it is nil
if
  i32.const 63
  call $writeCharLn                        ;; print '?'
else                                       ;; otherwise
  local.get $col
  i32.load
  i32.const 1                             ;; check if 'R'
  i32.eq
  if                                       ;; if it is 'R'
    i32.const 82
    call $writeCharLn                    ;; print 'R'
  else                                    ;; otherwise
    local.get $col
    i32.load
    i32.const 2                           ;; check if 'G'
    i32.eq
    if                                    ;; if it is 'G' kind
      i32.const 71
      call $writeCharLn                  ;; print 'G'
    else
      i32.const 63                       ;; otherwise, default
      call $writeCharLn                  ;; print '?'
    end
  end
end
end
```

# Memory Impact & Management

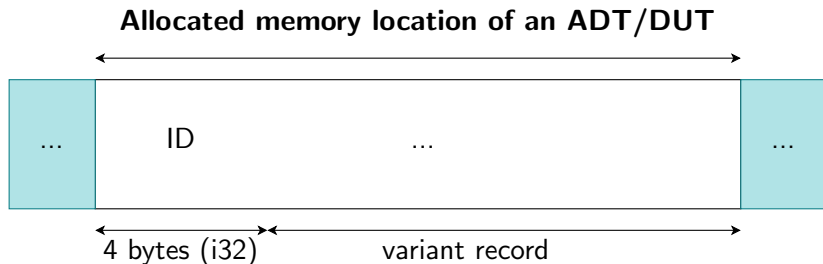
- Each instance of a DUT is located on the heap, and variables are just pointers to locations on the heap.





# Memory Impact & Management

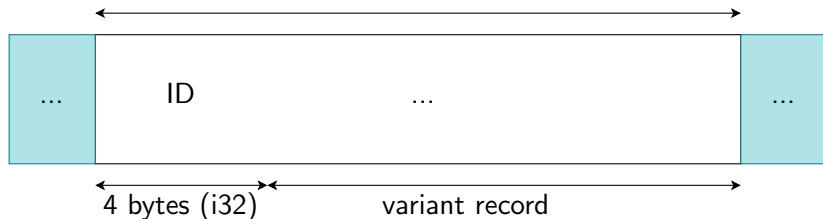
- Each instance of a DUT is located on the heap, and variables are just pointers to locations on the heap.
- Size of an allocation depends on the size of the variant being instantiated.



# Memory Impact & Management

- Each instance of a DUT is located on the heap, and variables are just pointers to locations on the heap.
- Size of an allocation depends on the size of the variant being instantiated.
- Offsets to accessing variables works the same as records, with a 4 byte offset for the variant id.

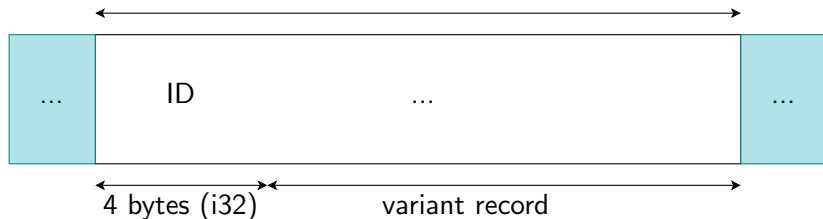
## Allocated memory location of an ADT/DUT



# Memory Impact & Management

- Each instance of a DUT is located on the heap, and variables are just pointers to locations on the heap.
- Size of an allocation depends on the size of the variant being instantiated.
- Offsets to accessing variables works the same as records, with a 4 byte offset for the variant id.
- Memory usage can be fairly inefficient for certain constructions (e.g., Strings, Lists, etc).

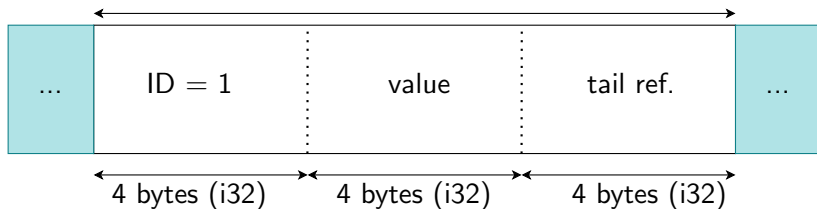
## Allocated memory location of an ADT/DUT



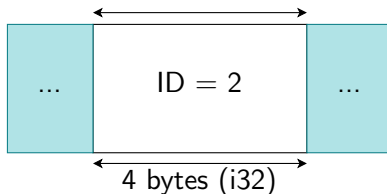
# Example: Integer Lists in Memory

Strings are very similar

## 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 always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.

# Notable Design Decisions

- The first 4 bytes of a program are always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.
- DUT variant/kind identifiers are immutable and indexed consecutively, starting from 1.

# Notable Design Decisions

- The first 4 bytes of a program are always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.
- DUT variant/kind identifiers are immutable and indexed consecutively, starting from 1.
- The data within the records are mutable, but you can only access them through caseing.

# Notable Design Decisions

- The first 4 bytes of a program are always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.
- DUT variant/kind identifiers are immutable and indexed consecutively, starting from 1.
- The data within the records are mutable, but you can only access them through caseing.
- When parsing DUTs, we create “instantiation helper functions” for each DUT variant.



# Notable Design Decisions

- The first 4 bytes of a program are always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.
- DUT variant/kind identifiers are immutable and indexed consecutively, starting from 1.
- The data within the records are mutable, but you can only access them through caseing.
- When parsing DUTs, we create “instantiation helper functions” for each DUT variant.
  - “DUT variant instantiation” is secretly rewritten into procedure calls.

# Notable Design Decisions

- The first 4 bytes of a program are always left as 0 so that we can always have uninitialized DUT pointers pointing to it naturally. When this uninitialized DUT is cased on, we will always see that the “instance” has  $id = 0$ , meaning it’s “nil”/not instantiated.
- DUT variant/kind identifiers are immutable and indexed consecutively, starting from 1.
- The data within the records are mutable, but you can only access them through caseing.
- When parsing DUTs, we create “instantiation helper functions” for each DUT variant.
  - “DUT variant instantiation” is secretly rewritten into procedure calls.
  - This simplifies parsing, and trims down generated code size.

# Example of DUT instantiation helper

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

```
(func $_mk.Cons (param $head i32) (param $tail i32) (result i32)
global.get $_memsize          ;; get known unused memory location
i32.const 1                    ;; get Cons's kind index
i32.store                      ;; store it
global.get $_memsize          ;; get known unused memory location
i32.const 4                    ;; get offset of the next type
i32.add                        ;; impose offset onto total memory size
local.get $head                ;; get param head
i32.store                      ;; store it in it's area
global.get $_memsize          ;; get known unused memory location
i32.const 8                    ;; get offset of the next type
i32.add                        ;; impose offset onto total memory size
local.get $tail                ;; get param tail
i32.store                      ;; store it in it's area
global.get $_memsize          ;; get global memory size
global.get $_memsize          ;; get global memory size (again)
i32.const 12                   ;; get size of kind (Cons)
i32.add                        ;; add to memory size
global.set $_memsize           ;; set memory size, leftover i32 on stack which is the
    returned pointer to the generated Cons
)
```

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

- 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.

- 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!

- 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.

# Challenges

- pywasm



# Challenges

- pywasm
  - wasmer!

# Challenges

- pywasm
  - wasmer!
- Recursive disjoint union types (self-references)

# Challenges

- pywasm
  - wasmer!
- Recursive disjoint union types (self-references)
  - Assume type existence while parsing

- pywasm
  - wasmer!
- Recursive disjoint union types (self-references)
  - Assume type existence while parsing
  - Instances are references to other memory locations

- pywasm
  - wasmer!
- Recursive disjoint union types (self-references)
  - Assume type existence while parsing
  - Instances are references to other memory locations
- Making the grammar reasonably “natural” feeling in P0

- core components  $\approx$  330 lines of code

- core components  $\approx$  330 lines of code
- supplementary components  $\approx$  450 lines of code

- core components  $\approx$  330 lines of code
- supplementary components  $\approx$  450 lines of code
- tests and documentation  $\approx$  1400 lines of code



- core components  $\approx$  330 lines of code
- supplementary components  $\approx$  450 lines of code
- tests and documentation  $\approx$  1400 lines of code
- $\sim$ 230 commits

- core components  $\approx$  330 lines of code
- supplementary components  $\approx$  450 lines of code
- tests and documentation  $\approx$  1400 lines of code
- $\sim$ 230 commits
- $\sim$ 18hrs for stable product

- core components  $\approx$  330 lines of code
- supplementary components  $\approx$  450 lines of code
- tests and documentation  $\approx$  1400 lines of code
- $\sim$ 230 commits
- $\sim$ 18hrs for stable product
- $\sim$ 42hrs for testing, debugging, documentation, and presentation slides

# Table of Contents

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation
- 4 Future Work**
- 5 Conclusion

- 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
  - 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
  - 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!

# Table of Contents

- 1 Objective & Implementation
- 2 Examples
- 3 Implementation Discussion & Evaluation
- 4 Future Work
- 5 Conclusion**



# Final Remarks

In conclusion,

# Final Remarks

In conclusion,

- Memory can grow quickly with recursive DUTs.

In conclusion,

- Memory can grow quickly with recursive DUTs.
- DUTs are easy to implement!

# Final Remarks

In conclusion,

- Memory can grow quickly with recursive DUTs.
- DUTs are easy to implement!
- DUTs are nice to work with!

- 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.