Higher Order Programming in Wybe

MCS "Work-In-Progress" Presentation

James Barnes (820946)

University of Melbourne

Semester 2, 2021

Supervisor: Peter Schachte

What is Wybe?

- Looks & feels imperative, yet programs declaratively
  - Interface integrity
  - Explicit data and control flow
- Numerous novel language features
  - Multiple inputs and outputs, with overloaded modes
  - Resources
  - Partial procedures (tests)
- Novel intermediate representation, LPVM
  - Currently lacks features present in other intermediate representations

Wybe Features – Overloaded Modes

- Allows for procedures to be "called in reverse"

Example (Overloaded modes)

```
def add( x:int,  y:int, ?z:int) { ?z = x + y }
def add( x:int, ?y:int,  z:int) { ?y = z - x }
def add(?x:int,  y:int,  z:int) { ?x = z - y }

?a = 1; ?b = 2
add(a, b, ?c)
add(a, ?d, c)
```

Wybe Features – Overloaded Modes

- Allows for procedures to be "called in reverse"

Example (Overloaded modes)

```
def add( x:int,  y:int, ?z:int) { ?z = x + y }
def add( x:int, ?y:int,  z:int) { ?y = z - x }
def add(?x:int,  y:int,  z:int) { ?x = z - y }

?a = 1; ?b = 2
add(a, b, ?c)
~~add(a, ?d, c)~~ ?d = b
```

Wybe Features – Resources

- A by-name parameter-passing mechanism

Example (Resource usage)

```
resource strings:list(string) = []

def collect(tree:tree(string)) use !strings {
    if { tree = node(?left, ?str, ?right) ::
            !collect(left)
            !collect(str)
            !collect(right)
    }
}

def collect(str:string) use !strings {
    ?strings = strings  ,, [str]
}
```

Wybe Features – Resources

- A by-name parameter-passing mechanism

Example (Resource usage, flattened)

```
def collect(tree:tree(string), strings0, ?strings)) {
    if { tree = node(?left, ?str, ?right) ::
            collect(left,  strings0, ?strings1)
            collect(str,   strings1, ?strings2)
            collect(right, strings2, ?strings)
       | else :: ?strings = strings0 }
}

def collect(str:string, strings0, ?strings) {
    ?strings = ?strings0 ,, [str]
}
```

Intermediate Representations & LPVM

A compiler's representation of a program for analysis and transformation.

Common intermediate representations each have their own problems:

- Name management problems, forward bias (impedes analyses)

Solutions?

- ϕ-nodes, σ-nodes, γ-nodes, ...
- These are band-aids, only adding to the complexity

LPVM takes a paradigm shift into logic programming:

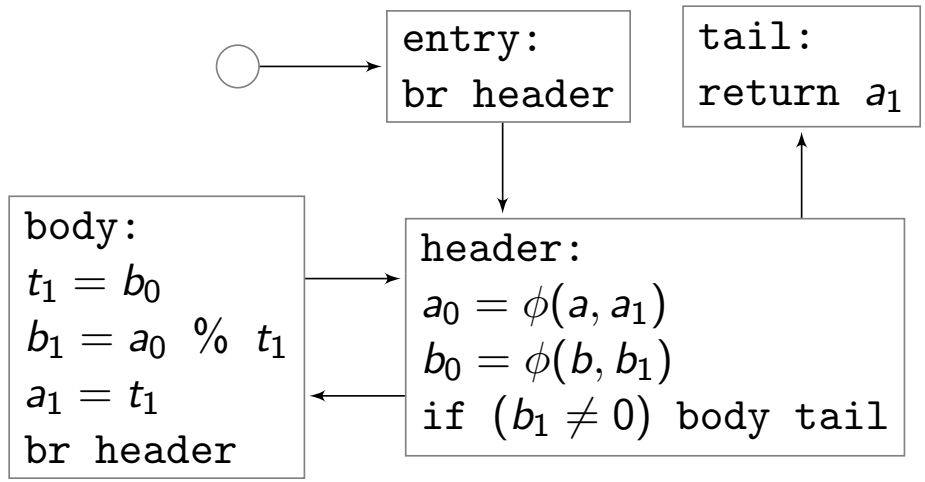
- Clauses are arranged such that non-determinism is tamed

LPVM vs SSA Form

Example:

```
int gcd(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % t;
        a = t;
    }
    return a;
}
```

SSA:



LPVM:

$$\texttt{gcd}(\texttt{a}, \texttt{b}, ?\texttt{r}) = \texttt{b} \neq 0 \ \wedge \ \texttt{mod}(\texttt{a}, \texttt{b}, ?\texttt{b}') \ \wedge \ \texttt{gcd}(\texttt{b}, \texttt{b}', ?\texttt{r})$$
$$\texttt{gcd}(\texttt{a}, \texttt{b}, ?\texttt{r}) = \texttt{b} = 0 \ \wedge \ ?\texttt{r} = \texttt{a}$$

What's Missing?

Higher order types! Unlike other language, Wybe does not support this feature.

Higher order types allow you to...

- Write succinct and more general code
- Pass procedures into and out of other procedures

LPVM, which used by the Wybe compiler, also does not support higher order types.

Research Questions

1. Can we extend the existing Wybe features to support higher order types?
2. How can we extend the LPVM implementation to support higher order types?
3. How does the existing Wybe implementation compare to the extended Wybe type system in terms of performance?

Research ~~Questions~~ Goals

1. Can we extend the existing Wybe features to support higher order types?
2. How can we extend the LPVM implementation to support higher order types?
3. How does the existing Wybe implementation compare to the extended Wybe type system in terms of performance?

Wybe Extension

We want to support higher order types in conjunction with the novel features of Wybe.

- Formalise, and extend, the Wybe type system to support higher order types
- Extending Wybe's features to support higher order types
  - Multiple modes, resources
  - Closures and anonymous functions
- Parsing new syntax, etc.

Higher Order Wybe

Example (List operations)

```
?list = [1, 2, 3, 4]

?sum = 0
for ?i in list {
    ?sum = sum + i
}
?prod = 1
for ?i in list {
    ?prod = prod * i
}

fold({ @ + @ = ?@ }, 0, list, ?sum)
fold({ @ * @ = ?@ }, 1, list, ?prod)
```

Higher Order Wybe

Example (Fold implementation)

```
def fold(f:(:?a, :?b, ?:?b), b0:?b, as:list(?a), ?b:?b) {
    ?b = b0
    for ?a in as {
        f(a, b, ?b)
    }
}
```

LPVM Extension

To support higher order types, the type system of LPVM must be extended.

- Formalisation of the existing LPVM type system
  - Typing rules, and proofs, that specify valid typings in LPVM
- Implementing higher order types within the formalised type system
- Various optimisations

Performance Comparison

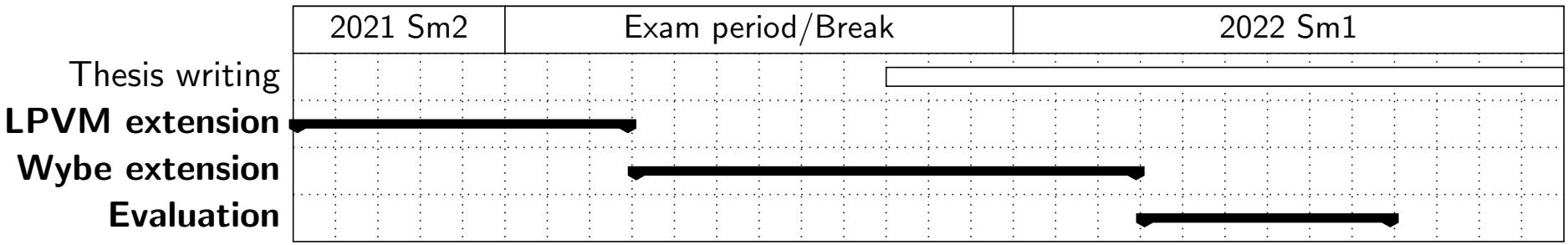
Ideally, higher order types should not cause major performance overheads.

- Creating higher order Wybe programs, transforming into first order
- Compare each compiled program pair
  - Compare execution time & code size

Hypothesis: higher order code will cause a slowdown.

Hypothesis: programs may be more succinct, but require more code generation.

Research Timeline



Thanks For Watching!

Any questions?