Assignment 2 Final Report of HasWasm: A WebAssembly-like Embedded DSL in Haskell

Faiz Ilham Muhammad (5231981) and Xinliang Lu (0822760)

30 June 2023

1 Introduction

HasWasm is a domain-specific language (DSL) that is integrated into Haskell. It makes use of the strong type system in Haskell to offer expressive and type-safe abstractions for writing WebAssembly code. HasWasm supports two basic types, blocks and control flows, functions, local and global variables, and export and imports. It also includes a code generator that turns the parsed Abstract Syntax Tree (AST) into a WebAssembly module in text (WAT) format.

This project is aimed to create a type-safe, expressive embedded DSL for WebAssembly programming in Haskell. Our effort is concentrated on offering expressive, type-safe, and user-friendly language abstractions for writing WebAssembly code.

2 Approach

2.1 HasWasm Overall Design

In designing the type signatures, we employ type families to define relationships between each types. We also want to avoid using any undecidable language extensions (e.g. UndecidableInstances) so that the compile time can be guaranteed to be terminating.

In HasWasm, a WebAssembly instruction's type is represented as a $TypedInstr$. For example, instruction $i32.add :: i32\ i32 \rightarrow i32$ is typed as follows.

```
i32_add :: (Stack s) => TypedInstr (s :+ I32 :+ I32) (s :+ I32)
```

This means that i32_add is an instruction that requires a Stack s with two I32 value on top of it, and will results in Stack s with only a single I32 value on top of it. Sequencing the instructions can be done easily using our defined # operator.

```
(#) :: (Stack a, Stack b, Stack c) => TypedInstr a b -> TypedInstr b c -> TypedInstr a c
```

A function is represented by the WasmFunc type:

```
myFunc :: WasmFunc p v r
```

in which p is the types of parameters, v is the types of local variables, and r is the types of returned values. Each of p, v and r can be a unit type, an I32, a F32, or tuple of I32 and/or F32 (up to 5 tuple members). Those types are grouped in a type family called VarTypes.

```
class BaseType t where ...
instance BaseType I32 where ...
instance BaseType F32 where ...

class VarTypes v where ...
instance VarTypes () where ...
instance VarTypes I32 where ...
instance VarTypes F32 where ...
instance (BaseType t1, BaseType t2) => VarTypes (t1, t2) where ...
instance (BaseType t1, BaseType t2, BaseType t3) => VarTypes (t1, t2, t3) where ...
...
```

To declare a function, users may use createLocalFunction or createExpFunction, which take a string for function name, and a function that takes a VarSet of parameters, a VarSet of local variables, a return instruction and returns the function body. The return instruction is provided by the createFunction so that it have the correct typing.

```
createExpFunction :: (VarTypes p, VarTypes v, VarTypes r) =>
  String -> (VarSet p -> VarSet v -> ReturnInstr (StackT s t) r -> FuncBody (StackT s t)
     r) -> WasmFunc p v r
```

In practice, users can declare a function as shown below.

```
-- takes 3 i32 parameter, have 1 i32 local, and return 1 i32
add :: WasmFunc (I32, I32) (I32) I32
add = createLocalFunction "rgb" func
  where
  -- Note: func type signature can be omitted for simplicity
  func :: (Stack s) => (Var I32, Var I32) -> Var I32 -> ReturnInstr s I32 -> FuncBody s
    I32
  func (a, b) l ret =
    local_get a #
    local_get b #
    i32_add #
    ret            -- no need explicit return here, but it can also use one
```

Blocks and loops use similar type family in their signatures. It takes a VarTypes of prerequisite types p, a VarTypes of resulting types r, and a function that takes the label of the block or loop, and returns the block body. StackType r s is equivalent to a type t_r^*, and FuncCallType s p r is equivalent to $t_p^* \to t_r^*$.

```
block :: (VarTypes p, VarTypes r, Stack s) => p -> r -> (TypedLabel (StackType r s) ->
    FuncCallType s p r) -> FuncCallType s p r

loop :: (VarTypes p, VarTypes r, Stack s) => p -> r -> (TypedLabel (StackType p s) ->
    FuncCallType s p r) -> FuncCallType s p r
```

In practice, users can use block, loops, and branching as such:

```
factorial :: WasmFunc I32 () I32
factorial = createExpFunction "factorial" func
  where
  func n () ret =
    i32_const 1 #
    block (I32) (I32) (\end-> -- end is the label id of the block
      loop (I32) (I32) (\start -> -- start is the label id of the loop
        local_get n #
        i32_const 1 #
        i32_le_s #
        br_if end #    -- if n <= 1, jump to end
        local_get n #
        i32_mul #      -- total = total * n
        local_get n #
        i32_const 1 #
        i32_sub #
        local_set n #  -- set n = n - 1
        br start       -- loop back
      )
    )
```

Build Implicitly Called Funcs and Global Vars By using the breadth-first search (BFS), this module recursively detects and automatically checks the following properties for a function or global variable:

- Being defined twice with different type signatures
- Being called recursively and implicitly

Then automatically adding the missing declarations for these functions and variables. Also, this module can prevent infinite nested calls.

2.2 Supported Wasm Instructions

HasWasm supports both basic i32 and f32 instructions, these include the following calculation operations[1]:

- $unop_{f32}$: neg, abs, ceil, floor, trunc, nearest, sqrt

[1] https://dl.acm.org/doi/pdf/10.1145/3062341.3062363

- $binop_{i32}$: add, sub, mul, div_s, div_u, rem_s, rem_u, and, or, xor, shl, shr_s, shr_u, rotl, rotr
- $binop_{f32}$: add, sub, mul, div, min, max, copysign
- $testop_{i32}$: eqz
- $relop_{i32}$: eq, ne, lt_s, lt_u, gt_s, gt_u, le_s, le_u, ge_s, ge_u
- $relop_{f32}$: eq, ne, lt, gt, le, ge

and other instructions1:

- $i32.const$, $f32.const$
- block tf e* end
- loop tf e* end
- br i, br_if i
- call i, return
- get_local i, set_local i
- get_global i, set_global i

3 Results and Discussion

In this section, we will show how to use HasWasm and discuss its features.

3.1 Accomplished Functionalities

- Utilizing type families and Generalized Algebraic Data Types (GADT) to guarantee type-safety.
- Generating WebAssembly code in text format.
- Supporting basic i32 & f32 instructions, blocks and control flows (if-else, jumps), making and declaring functions & local variables & global variables, exporting and importing functions.

3.2 Example

Here we provide an example for using HasWasm. Users first declare exported functions using "createExpFunction", or local functions using "createLocalFunction":

```
-- an exported factorial function, that takes an i32 and returns i32
factorial :: WasmFunc I32 () I32
factorial = createExpFunction "factorial" func
  where
  func n () ret =
    i32_const 1 #
    block (I32) (I32) (\end->
      loop (I32) (I32) (\start ->
        local_get n #
        i32_const 1 #
        i32_le_s #
        br_if end #     -- if n <= 1, jump to end
        local_get n #
        i32_mul #       -- total = total * n
        local_get n #
        i32_const 1 #
        i32_sub #
        local_set n #  -- set n = n - 1
        br start        -- loop back
      )
    )

-- recursive version of the factorial function called "factorial_rec"
-- it is also exported and takes an i32 and returns i32
factorialRec :: WasmFunc I32 () I32
```

```
factorialRec = createExpFunction "factorial_rec" func
  where
  func n _ ret =
    block () () (\lbl ->
      local_get n #
      br_if lbl #
      i32_const 1 #
      ret             -- return 1 if n == 0
    ) #
    local_get n #
    local_get n #
    i32_const 1 #
    i32_sub #
    call factorialRec #
    i32_mul

-- an exported function that increase a global "counter" variable
incCounter :: WasmFunc () () ()
incCounter = createExpFunction "inc_counter" func
  where
  func _ _ _ =
    global_get counter #
    call addByConst #
    global_set counter #
    global_get counter #
    call consoleLog

-- a local function that increase the given argument
-- with some value in the "constant" global variable
addByConst :: WasmFunc I32 () I32
addByConst = createLocalFunction "addByConst" func
  where
  func i _ _ =
    local_get i #
    global_get constant #
    i32_add
```

Users can also declare global variables and imported functions:

```
-- a mutable global variable of type i32
counter :: GlobalVar Mut I32
counter = createGlobalI32 "counter" (Just "counter") 0

-- an immutable global variable of type i32
constant :: GlobalVar Imm I32
constant = createGlobalI32 "constant" Nothing 1

-- an import function named "console_log that takes i32 and returns nothing
-- the function should be provided with console.log by the host of the wasm code
consoleLog :: WasmFunc I32 () ()
consoleLog = createImportFunction "console" "log" "console_log"
```

Finally, users can make a module that includes all the declared functions and global variables. Functions or global variables that are not added manually by the user, but are called indirectly by one of the included functions will also be included automatically into the module. In this example, "constant", "counter", "addByConst", and "consoleLog" will be automatically included because "incCounter" directly or indirectly uses them.

```
myModule :: WasmModule
myModule = createModule $ do
  addFunc factorial
  addFunc factorialRec
  addFunc incCounter
```

Taken together, the full program would look like this:

```
module Main where

import HasWasm
import HasWasm.Instruction

main :: IO ()
main = do
  case buildModule myModule of
```

```
    Right result -> putStrLn $ result
    Left err -> putStrLn $ "Error: " ++ err

myModule :: WasmModule
myModule = createModule $ do
  addFunc factorial
  addFunc factorialRec
  addFunc incCounter

-- an exported factorial function, that takes an i32 and returns i32
factorial :: WasmFunc I32 () I32
factorial = createExpFunction "factorial" func
  where
  func n () ret =
    i32_const 1 #
    block (I32) (I32) (\end->
      loop (I32) (I32) (\start ->
        local_get n #
        i32_const 1 #
        i32_le_s #
        br_if end #    -- if n <= 1, jump to end
        local_get n #
        i32_mul #      -- total = total * n
        local_get n #
        i32_const 1 #
        i32_sub #
        local_set n #  -- set n = n - 1
        br start       -- loop back
      )
    )

-- recursive version of the factorial function called "factorial_rec"
-- it is also exported and takes an i32 and returns i32
factorialRec :: WasmFunc I32 () I32
factorialRec = createExpFunction "factorial_rec" func
  where
  func n _ ret =
    block () () (\lbl ->
      local_get n #
      br_if lbl #
      i32_const 1 #
      ret            -- return 1 if n == 0
    ) #
    local_get n #
    local_get n #
    i32_const 1 #
    i32_sub #
    call factorialRec #
    i32_mul

-- an exported function that increase a global "counter" variable
incCounter :: WasmFunc () () ()
incCounter = createExpFunction "inc_counter" func
  where
  func _ _ _ =
    global_get counter #
    call addByConst #
    global_set counter #
    global_get counter #
    call consoleLog

-- a local function that increase the given argument
-- with some value in the "constant" global variable
addByConst :: WasmFunc I32 () I32
addByConst = createLocalFunction "addByConst" func
  where
  func i _ _ =
    local_get i #
    global_get constant #
    i32_add

-- a mutable global variable of type i32
counter :: GlobalVar Mut I32
```

```
counter = createGlobalI32 "counter" (Just "counter") 0

-- an immutable global variable of type i32
constant :: GlobalVar Imm I32
constant = createGlobalI32 "constant" Nothing 1

-- an import function named "console_log that takes i32 and returns nothing
-- the function should be provided with console.log by the host of the wasm code
consoleLog :: WasmFunc I32 () ()
consoleLog = createImportFunction "console" "log" "console_log"
```

When it runs, the program will produce the following WebAssembly code in text format:

```
(module
  (import "console" "log" (func $console_log (param i32) ))
  (export "counter" (global $counter))
  (export "factorial" (func $factorial))
  (export "factorial_rec" (func $factorial_rec))
  (export "inc_counter" (func $inc_counter))
  (func $addByConst (param i32) (result i32)
    local.get 0
    global.get $constant
    i32.add
  )
  (global $constant i32 (i32.const 1))
  (global $counter (mut i32) (i32.const 0))
  (func $factorial (param i32) (result i32)
    i32.const 1
    (block $l0 (param i32) (result i32)
      (loop $l1 (param i32) (result i32)
        local.get 0
        i32.const 1
        i32.le_s
        br_if $l0
        local.get 0
        i32.mul
        local.get 0
        i32.const 1
        i32.sub
        local.set 0
        br $l1
      )
    )
  )
  (func $factorial_rec (param i32) (result i32)
    (block $l0
      local.get 0
      br_if $l0
      i32.const 1
      return
    )
    local.get 0
    local.get 0
    i32.const 1
    i32.sub
    call $factorial_rec
    i32.mul
  )
  (func $inc_counter
    global.get $counter
    call $addByConst
    global.set $counter
    global.get $counter
    call $console_log
  )
)
```

3.3 HasWasm Evaluation

We evaluate HasWasm by answering the following questions:

Does the DSL ensure the type correctness of the constructed Wasm code? The answer is yes, the sequencing operator (#) and the type families for functions and block ensures that the generated WASM code will be type safe. However, our types are currently more strict than the WebAssembly specification, in particular for br and return. For example the specification defines that, for $C(i) = t^*$, the typing judgment of br is $C \vdash \mathbf{br}\ i : t_1^* t^* \rightarrow t_2^*$, while our implementation of br instruction has the equivalent of typing judgment $C \vdash \mathbf{br}\ i : t^* \rightarrow t_2^*$.

Is the syntax similar enough to wasm text syntax? The answer is yes, in the above code example, we show that HasWasm provides a similar way to directly manipulate the stack like WebAssembly but using GADT to ensure the type-safety.

4 Limitations

There are several limitations in our project that we couldn't implement due to limited time:

1. Functions can only have at maximum 5 parameters, 5 local variables, and 5 return values. The same limitations also apply for block and loop prerequisite and resulting type.
2. The type signature for br and return is more restrictive than the specification, meaning that there are programs that should be allowed but will be rejected by our DSL.
3. Name clash checking of functions and global variables are done in runtime, not compile time.
4. Code optimizer and binary-code generator are not implemented.

5 Conclusion

We have developed HasWasm, an embedded DSL for writing WebAssembly code in Haskell. The project's primary goals, which included developing a type-safe and expressive tool for Wasm code generation, were met. Future development can be focused on fixing the limitations in our implementation.