P0 Code Generator for WASM

Emil Sekerinski, McMaster University, revised February 2022

The generated code is kept in memory and all code generation procedures continuously append to that code: procedure `genProgStart` initializes the generator, then `gen` -prefixed procedures are to be called for P0 constructs in the same order in which they are recognized by a recursive descent parser, and finally procedure `genProgExit` returns the generated code in assembly language as a string in textual WebAssembly. The generation procedures are:

- `genBool` , `genInt` , `genRec` , `genArray` , `genSet`
- `genProgStart` , `genGlobalVars` , `genProgEntry` , `genProgExit`
- `genProcStart` , `genLocalVars` , `genProcEntry` , `genProcExit`
- `genIndex` , `genSelect` , `genVar` , `genConst` , `genUnaryOp` , `genBinaryOp` , `genRelation`
- `genLeftAssign` , `genRightAssign` , `genAssign` , `genActualPara` , `genCall`
- `genRead` , `genWrite` , `genWriteln`
- `genSeq` , `genThen` , `genIfThen` , `genElse` , `genIfElse` , `genWhile` , `genDo` , `genWhileDo`

Errors in the code generator are reported by calling `mark` of the scanner. The data types of the symbol table are used to specify the P0 constructs for which code is to be generated.

In [ ]:

```python
import nbimporter; nbimporter.options["only_defs"] = False
from SC import TIMES, DIV, MOD, PLUS, MINUS, AND, OR, EQ, NE, LT, GT, LE, GE, \
     NOT, CARD, COMPLEMENT, UNION, INTERSECTION, ELEMENT, SUBSET, SUPERSET, SET, \
     mark
from ST import indent, Var, Const, Type, Proc, StdProc, Int, Bool, Array, \
     Record, Set
```

Following variables determine the state of the code generator:

- `curlev` is the current level of nesting of P0 procedures
- `memsize` is the size of the memory, in which records and arrays are allocated
- `asm` is a list of strings with the WASM instruction in textual form

Procedure `genProgStart()` initializes these variables.

In [ ]:

```python
def genProgStart():
    global curlev, memsize, asm
    curlev, memsize = 0, 0
    asm = ['(module',
           '(import "P0lib" "write" (func $write (param i32)))',
           '(import "P0lib" "writeln" (func $writeln))',
           '(import "P0lib" "read" (func $read (result i32)))']
```

Following procedures "generate code" for all P0 types by determining the size of objects and store in the `size` field.

- Integers and booleans occupy 4 bytes
- The size of a record is the sum of the sizes of its field; the offset of a field is the sum of the size of the preceding fields
- The size of an array is its length times the size of the base type.

In [ ]:

```python
def genBool(b: Bool):
    b.size = 1; return b

def genInt(i: Int):
    i.size = 4; return i

def genRec(r: Record):
    s = 0
    for f in r.fields:
        f.offset, s = s, s + f.tp.size
    r.size = s
    return r

def genArray(a: Array):
    a.size = a.length * a.base.size
    return a

def genSet(s: Set):
    if s.lower < 0 or s.lower + s.length > 32:
        mark('WASM: set too large')
    s.size = 4; return s
```

The symbol table assigns to each entry the level of declaration in the field `lev: int` . Variables are assigned a `name: str` field by the symbol table and an `adr: int` field by the code generator. The use of the `lev` field is extended:

In [ ]:

```
Global = 0; Stack = -1; MemInd = -2; MemAbs = -3
```

- `lev > 0` : local `Int` , `Bool` , `Set` variable or parameter allocated in the procedure (function) call frame, accessed by `name` or `Array` , `Record` variable in memory with address in local variable `name` .
- `lev = Global` : global `Int` , `Bool` , `Set` variable allocated as a WebAssembly global variable, accessed by `name` ,
- `lev = Stack` : `Int` , `Bool` , `Set` variable on the expression stack, `Array` , `Record` variable in memory with address on the expression stack,
- `lev = MemInd` : `Int` , `Bool` , `Set` variable in WebAssembly memory with address on expression stack
- `lev = MemAbs` : `Int` , `Bool` , `Set` , `Array` , `Record` variable allocated in WebAssembly memory, accessed by `adr` .

For each declared global variable, `genGlobalVars(sc, start)` allocates a global WebAssembly variable by the same name, if the type is `Int` or `Bool` , or reserves space in the memory, if the type is `Array` , `Record` . The parameter `sc` contains the top scope with all declarations parsed so far; only variable declarations from index `start` on in the top scope are considered.

In [ ]:

```
def genGlobalVars(sc, start):
    global memsize
    for i in range(start, len(sc)):
        if type(sc[i]) == Var:
            if sc[i].tp in (Int, Bool) or type(sc[i].tp) == Set:
                asm.append('(global $' + sc[i].name + ' (mut i32) i32.const 0)')
            elif type(sc[i].tp) in (Array, Record):
                sc[i].lev, sc[i].adr, memsize = MemAbs, memsize, memsize + sc[i].tp.size
            else: mark('WASM: type?')

def genLocalVars(sc, start):
    for i in range(start, len(sc)):
        if type(sc[i]) == Var:
            asm.append('(local $' + sc[i].name + ' i32)')
    asm.append('(local $0 i32)') # auxiliary local variable
    return sc[start:]
```

Procedure `loadItem(x)` generates code for loading `x` on the expression stack, assuming `x` is global `Var` , local `Var` , stack `Var` , memory `Var` , or `Const` .

In [ ]:

```
def loadItem(x):
    if type(x) == Var:
        if x.lev == Global: asm.append('global.get $' + x.name) # global Var
        elif x.lev == curlev: asm.append('local.get $' + x.name) # local Var
        elif x.lev == MemInd: asm.append('i32.load')
        elif x.lev == MemAbs:
            asm.append('i32.const ' + str(x.adr))
            if x.tp in {Int, Bool}: asm.append('i32.load')
        elif x.lev != Stack: mark('WASM: var level!') # already on stack if lev == Stack
    else: asm.append('i32.const ' + str(x.val))
```

In [ ]:

```
def genVar(x):
    if Global < x.lev < curlev: mark('WASM: level!')
    y = Var(x.tp); y.lev, y.name = x.lev, x.name
    if x.lev == MemAbs: y.adr = x.adr
    return y
```

Procedure `genConst(x)` does not need to generate any code.

In [ ]:

```
def genConst(x):
    # x is Const
    x.lev = None # constants are either not stored or on stack, lev == Stack
    return x
```

Procedure `genUnaryOp(op, x)` generates code for `op x` if `op` is `MINUS` , `NOT` , `CARD` , `COMPLEMENT` , `Set` ; the `Set` operation is for a singleton set. If `op` is `AND` , `OR` , item `x` is the first operand and an `if` instruction is generated.

In [ ]:

```
def genUnaryOp(op, x):
    loadItem(x)
    if op == MINUS:
        asm.append('i32.const -1')
        asm.append('i32.mul')
        x = Var(Int); x.lev = Stack
    elif op == CARD:
        asm.append('i32.popcnt')
        x = Var(Int); x.lev = Stack
    elif op == COMPLEMENT:
        u = (1 << x.tp.length) - 1 # x.tp.length 1's
        u = u << x.tp.lower # universe of base type
        asm.append('i32.const ' + hex(u))
        asm.append('i32.xor')
        x = Var(x.tp); x.lev = Stack
    elif op == SET:
        asm.append('local.set $0')
        asm.append('i32.const 1')
        asm.append('local.get $0')
        asm.append('i32.shl')
        x = Var(Set(0, 32)); x.lev = Stack
    elif op == NOT:
        asm.append('i32.eqz')
        x = Var(Bool); x.lev = Stack
    elif op == AND:
        asm.append('if (result i32)')
        x = Var(Bool); x.lev = Stack
    elif op == OR:
        asm.append('if (result i32)')
        asm.append('i32.const 1')
        asm.append('else')
        x = Var(Bool); x.lev = Stack
    elif op == ELEMENT:
        asm.append('local.set $0')
        asm.append('i32.const 1')
        asm.append('local.get $0')
        asm.append('i32.shl')
        x = Var(Int); x.lev = Stack
    elif op in {SUBSET, SUPERSET}:
        asm.append('local.tee $0')
        asm.append('local.get $0')
        x.lev = Stack
    else: mark('WASM: unary operator?')
    return x
```

Procedure `genBinaryOp(op, x, y)` generates code for `x op y` if `op` is `PLUS`, `MINUS`, `TIMES`, `DIV`, `MOD`, `UNION`, `INTERSECTION`. If `op` is `AND`, `OR`, code for `x` and the start of an `if` instruction has already been generated; code for `y` and the remainder of the `if` instruction is generated.

In [ ]:

```
def genBinaryOp(op, x, y):
    if op in (PLUS, MINUS, TIMES, DIV, MOD):
        loadItem(x); loadItem(y)
        asm.append('i32.add' if op == PLUS else \
                   'i32.sub' if op == MINUS else \
                   'i32.mul' if op == TIMES else \
                   'i32.div_s' if op == DIV else \
                   'i32.rem_s' if op == MOD else '?')
        x = Var(Int); x.lev = Stack
    elif op in {UNION, INTERSECTION}:
        loadItem(x); loadItem(y)
        asm.append('i32.or' if op == UNION else \
                   'i32.and' if op == INTERSECTION else '?')
        x = Var(x.tp); x.lev = Stack
    elif op == AND:
        loadItem(y) # x is already on the stack
        asm.append('else')
        asm.append('i32.const 0')
        asm.append('end')
        x = Var(Bool); x.lev = Stack
    elif op == OR:
        loadItem(y) # x is already on the stack
        asm.append('end')
        x = Var(Bool); x.lev = Stack
    else: mark('WASM: binary operator?')
    return x
```

Procedure `genRelation(op, x, y)` generates code for `x op y` if `op` is `EQ`, `NE`, `LT`, `LE`, `GT`, `GE`, `ELEMENT`, `SUBSET`, `SUPERSET`.

In [ ]:

```
def genRelation(op, x, y):
    loadItem(x); loadItem(y)
    asm.extend(['i32.eq'] if op == EQ else \
               ['i32.ne'] if op == NE else \
               ['i32.lt_s'] if op ==  LT else \
               ['i32.gt_s'] if op == GT else \
               ['i32.le_s'] if op == LE else \
               ['i32.ge_s'] if op == GE else \
               ['i32.and'] if op == ELEMENT else \
               ['i32.and', 'i32.eq'] if op == SUBSET else \
               ['i32.or', 'i32.eq'] if op == SUPERSET else '?')
    x = Var(Bool); x.lev = Stack
    return x
```

Procedure `genIndex(x, y)` generates code for `x[y]`, assuming `x` is `Var` or `Ref`, `x.tp` is `Array`, and `y.tp` is `Int`. If `y` is `Const`, only `x.adr` is updated and no code is generated, otherwise code for array index calculation is generated.

In [ ]:

```
def genIndex(x, y):
    # x[y], assuming x.tp is Array and x is global Var, local Var
    # and y is Const, local Var, global Var, stack Var
    if x.lev == MemAbs and type(y) == Const:
        x.adr += (y.val - x.tp.lower) * x.tp.base.size
        x.tp = x.tp.base
    else:
        loadItem(y)
        if x.tp.lower != 0:
            asm.append('i32.const ' + str(x.tp.lower))
            asm.append('i32.sub')
        asm.append('i32.const ' + str(x.tp.base.size))
        asm.append('i32.mul')
        if x.lev > 0: asm.append('local.get $' + x.name)
        elif x.lev == MemAbs: asm.append('i32.const ' + str(x.adr))
        asm.append('i32.add')
        x = Var(x.tp.base)
        if x.tp in (Int, Bool) or type(x.tp) == Set: x.lev = MemInd
        else: x.lev = Stack
    return x
```

Procedure `genSelect(x, f)` generates code for `x.f`, provided `f` is in `x.fields`. If `x` is `Var`, i.e. allocated in memory, only `x.adr` is updated and no code is generated. If `x` is `Ref`, i.e. a reference to memory, code for adding the offset of `f` is generated. An updated item is returned.

In [ ]:
```
def genSelect(x, f):
    # x.f, assuming x.tp is Record, f is Field, and x.lev is Stack, MemInd or is > 0
    if x.lev == MemAbs: x.adr += f.offset
    elif x.lev == Stack:
        asm.append('i32.const ' + str(f.offset))
        asm.append('i32.add')
    elif x.lev > 0:
        asm.append('local.get $' + x.name) # parameter or local reference
        asm.append('i32.const ' + str(f.offset))
        asm.append('i32.add')
        x.lev = Stack
    else: mark('WASM: select?')
    x.tp = f.tp
    return x
```

Procedures `genLeftAssign` and `genRightAssign` prepare for code generation for multiple assignment statements: `genLeftAssign` completes the generated code for a variable and designator on the left-hand side of an assignment statement. That is only needed for an array that is indexed with a constant expression, as in that case no code was generated.

In [ ]:
```
def genLeftAssign(x):
    if x.lev == MemAbs: asm.append('i32.const ' + str(x.adr))
    elif x.lev > 0 and type(x.tp) in (Array, Record):
        asm.append('local.get $' + x.name)
    return x
```

Procedure `genRightAssign` generated code that pushes the right-hand side of an assignment onto the stack, if it is not already there.

In [ ]:
```
def genRightAssign(x):
    loadItem(x); y = Var(x.tp); y.lev = Stack; return y
```

Procedure `genAssign(x, y)` generates code for `x := y`, provided `x` is `Var`, `Ref` and `y` is `Var`, `Ref`. The procedure assumes in case `x` refers to an array element, the address is already on the stack and that `y` is on the stack.

In [ ]:
```
def genAssign(x, y):
    loadItem(y)
    if x.lev == Global: asm.append('global.set $' + x.name)
    elif x.lev > 0:
        if type(x.tp) in (Array, Record):
            asm.append('i32.const ' + str(x.tp.size))
            asm.append('memory.copy')
        else: asm.append('local.set $' + x.name)
    else:
        if type(x.tp) in (Array, Record):
            asm.append('i32.const ' + str(x.tp.size))
            asm.append('memory.copy')
        else: asm.append('i32.store')
```

In [ ]:
```
def genProgEntry(ident):
    global curlev
    curlev = curlev + 1
    asm.append('(global $_memsize (mut i32) i32.const ' + str(memsize) + ')')
    asm.append('(func $program')

def genProgExit(x):
    global curlev
    curlev = curlev - 1
    asm.append('(memory ' + str(memsize // 2** 16 + 1) + ')\n(start $program)\n)')
    return '\n'.join(l for l in asm)

def genProcStart(ident, fp, rp):
    global curlev
    if curlev > 0: mark('WASM: no nested procedures')
    curlev = curlev + 1
    asm.append('(func $' + ident + ' ' +
               ' '.join('(param $' + e.name + ' i32)' for e in fp) + ' ' +
               ' '.join('(result i32)' for e in rp) +
               ('\n' if len(rp) > 0 else '') +
               '\n'.join('(local $' + e.name + ' i32)' for e in rp))
    return rp

def genProcEntry(ident, para, local):
    pl = (para if para else []) + local
    if any(type(l) == Var and type(l.tp) in (Array, Record) for l in pl):
        asm.append('(local $_fp i32)')
        asm.append('global.get $_memsize')
        asm.append('local.set $_fp')
    for l in pl:
        if type(l) == Var and type(l.tp) in (Array, Record):
            asm.append('global.get $_memsize')
            asm.append('i32.const ' + str(l.tp.size))
            asm.append('i32.add')
            asm.append('local.tee $' + l.name)
            asm.append('global.set $_memsize')

def genProcExit(x, para, local):
    global curlev
    curlev = curlev - 1
    if any(type(l) == Var and type(l.tp) in (Array, Record) for l in local):
        asm.append('local.get $_fp')
        asm.append('global.set $_memsize')
    if para: asm.append('\n'.join('local.get $' + e.name for e in para))
    asm.append(')')

def genActualPara(ap, fp, n):
    if ap.tp in {Int, Bool} or type(ap.tp) == Set: loadItem(ap)
    else: # a.tp is Array, Record
        if ap.lev > 0: asm.append('local.get $' + ap.name)
        elif ap.lev == MemAbs: asm.append('i32.const ' + str(ap.adr))
        elif ap.lev != Stack: mark('WASM: actual parameter?')

def genCall(rp, pr, ap): # result (or None), procedure, actual parameters
    asm.append('call $' + pr.name)
    for r in reversed(rp): y = Var(Int); y.lev = Stack; genAssign(r, y)

def genRead(x):
    asm.append('call $read')
    y = Var(Int); y.lev = Stack; genAssign(x, y)

def genWrite(x):
    asm.append('call $write')

def genWriteln():
    asm.append('call $writeln')
```

In [ ]:

```
def genSeq(x, y):
    pass

def genThen(x):
    loadItem(x)
    asm.append('if')
    return x

def genIfThen(x, y):
    asm.append('end')

def genElse(x, y):
    asm.append('else')

def genIfElse(x, y, z):
    asm.append('end')

def genWhile():
    asm.append('loop')

def genDo(x):
    loadItem(x)
    asm.append('if')
    return x
```

```
def genWhileDo(t, x, y):
    asm.append('br 1')
    asm.append('end')
    asm.append('end')
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js