計算機科学実験3 最終レポート

杉本風斗

平成 28 年 5 月 27 日

1 概要

動作は Small C の動作仕様に準じている.
make で実行プログラムをビルドし, 引数にソースファイルを渡すことで MIPS コードを生成する.

```
$ make
$ ./small-c program.sc
```

また, make test でユニットテストや spim エミュレータを用いた統合テストを実行できるように工夫した. テスト実行時にコードカバレッジも測定される.

```
$ make test
```

2 課題 11: データフロー解析

到達可能定義解析を実装した.
データフロー解析と最適化の処理は, optimize.go の Optimize() 関数に実装している.

以下にコードを示す. 簡単な説明をコード中のコメントにつけた.

```
// optimize.go
type DataflowBlock struct {
    Name       string  // BEGIN, END
    Statements []IRStatement
    Next       []*DataflowBlock
    Prev       []*DataflowBlock
}

func Optimize(program *IRProgram) *IRProgram {
    for i, f := range program.Functions {
        statements := flatStatement(f)

        // 中間表現プログラム列をデータフローのブロックごとに分ける
```

```
        blocks := splitStatementsIntoBlocks(statements)

        // ブロックの配列からデータフローを構成
        // block それぞれについて，block.Next を設定していく
        buildDataflowGraph(blocks)

        // データフローを見て不動点反復により到達可能定義解析する
        // 返り値はブロックごとに，各シンボルの到達可能な定義文 を入れた map
        // blockOut = (DataflowBlock -> (*Symbol -> []IRStatement))
        blockOut := searchReachingDefinitions(blocks)

        // ...
    }

    return program
}

// 不動点反復なので、状態が収束するまで地道に解析して状態を更新していくという雰囲気
func searchReachingDefinitions(blocks []*DataflowBlock) map[*DataflowBlock]BlockState {
    blockOut := make(map[*DataflowBlock]BlockState)

    changed := true
    for changed {
        changed = false

        for _, block := range blocks {
            inState := analyzeBlock(blockOut, block)
            if !inState.Equal(blockOut[block]) {
                changed = true
            }

            blockOut[block] = inState
        }
    }

    return blockOut
}

// ひとつのプログラム点を見て状態を更新する
// 到達可能定義解析の実質的な処理
func analyzeReachingDefinition(statement IRStatement, inState BlockState) BlockState {
    switch s := statement.(type) {
    case *IRAssignmentStatement:
```

```
            inState[s.Var] = []IRStatement{s}
            symbols := extractAddressVarsFromExpression(s.Expression)
            for _, symbol := range symbols {
                inState[symbol] = append(inState[symbol], s)
            }

        case *IRReadStatement:
            inState[s.Dest] = []IRStatement{s}

        // ポインタ参照書き込みがあったら，諦めムードにしておく
        case *IRWriteStatement:
            for symbol := range inState {
                inState[symbol] = append(inState[symbol], s)
            }

        case *IRCallStatement:
            inState[s.Dest] = []IRStatement{s}

        }

        return inState
}
```

3 課題 12: 最適化

定数畳み込みと無駄な命令の除去を実装した.

両方とも, 到達可能定義解析で得た情報を利用して実装している.

```
func Optimize(program *IRProgram) *IRProgram {
    for i, f := range program.Functions {
        // ...
        blockOut := searchReachingDefinitions(blocks)

        // 実装の都合で文ごとの到達可能定義を計算しなおしている
        allStatementState := reachingDefinitionsOfStatements(blocks, blockOut, statements)

        // 定数畳み込み
        program.Functions[i] = transformByConstantFolding(program.Functions[i], allStatementState
        // 無駄コード除去
        program.Functions[i] = transformByDeadCodeElimination(program.Functions[i], allStatementS
    }
```

```
    return program
}
```

3.1 定数畳み込み

中間表現の代入文に対して, 再帰的に定数畳み込みを行う. オペランドが両方とも定数の演算子を発見した場合, 直接計算結果を埋め込む.

以下にコードを示す.

```
func transformByConstantFolding(f *IRFunctionDefinition, allStatementState map[IRStatement]BlockS
    traversed := Traverse(f, func(statement IRStatement) IRStatement {
        foldConstantStatement(statement, allStatementState)
        return statement
    })

    return traversed.(*IRFunctionDefinition)
}

// 代入文なら expression を見て、それが定数だったら埋め込む
func foldConstantStatement(statement IRStatement, allStatementState map[IRStatement]BlockState) (
    switch s := statement.(type) {
    case *IRAssignmentStatement:
        isConstant, value := foldConstantExpression(s, s.Expression, allStatementState)
        if isConstant {
            s.Expression = &IRNumberExpression{Value: value}
            return true, value
        }
    }

    return false, 0
}

// 到達可能定義の情報を使って、再帰的に定数畳み込みしていく
func foldConstantExpression(statement IRStatement, expression IRExpression, allStatementState map
    switch e := expression.(type) {
    case *IRNumberExpression:
        return true, e.Value

    case *IRVariableExpression:
        symbol := e.Var
        definitionOfVar := allStatementState[statement][symbol]
        if len(definitionOfVar) == 1 && definitionOfVar[0] != statement {
```

```
            return foldConstantStatement(definitionOfVar[0], allStatementState)
        }

        return false, 0

    case *IRBinaryExpression:
        leftIsConstant, leftValue := foldConstantExpression(statement, e.Left, allStatementState)
        rightIsConstant, rightValue := foldConstantExpression(statement, e.Right, allStatementSta

        if leftIsConstant {
            e.Left = &IRNumberExpression{Value: leftValue}
        }

        if rightIsConstant {
            e.Right = &IRNumberExpression{Value: rightValue}
        }

        if leftIsConstant && rightIsConstant {
            switch e.Operator {
            case "+":
                return true, leftValue + rightValue

            case "-":
                return true, leftValue - rightValue

            case "*":
                return true, leftValue * rightValue

            case "/":
                return true, leftValue / rightValue

            case "<":
                value := 0
                if leftValue < rightValue {
                    value = 1
                }
                return true, value

            case ">":
                value := 0
                if leftValue > rightValue {
                    value = 1
                }
```

```
                return true, value

            case "<=":
                value := 0
                if leftValue <= rightValue {
                    value = 1
                }
                return true, value

            case ">=":
                value := 0
                if leftValue >= rightValue {
                    value = 1
                }
                return true, value

            case "==":
                value := 0
                if leftValue == rightValue {
                    value = 1
                }
                return true, value

            case "!=":
                value := 0
                if leftValue != rightValue {
                    value = 1
                }
                return true, value

            }

            panic("unexpected operator: " + e.Operator)
        }

        return false, 0
    }

    return false, 0
}
```

3.2 無駄な命令の除去

文を使用しているかどうかを到達可能定義を用いて記録し, 無駄な文を発見したら消す操作を収束するまで繰り返す. 無駄な命令を除去した結果, 不要になった変数宣言を最後に削除している.

```
func transformByDeadCodeElimination(f *IRFunctionDefinition, allStatementState map[IRStatement]Bl
    changed := true
    for changed {
        changed = false

        used := make(map[IRStatement]bool)
        markAsUsed := func(s IRStatement, symbol *Symbol) {
            for _, definition := range allStatementState[s][symbol] {
                used[definition] = true
            }
        }

        Traverse(f, func(statement IRStatement) IRStatement {
            switch s := statement.(type) {
            case *IRCompoundStatement:
                used[s] = true

            case *IRAssignmentStatement:
                if s.Var.IsGlobal() {
                    used[s] = true
                }

                vars := extractVarsFromExpression(s.Expression)
                for _, v := range vars {
                    markAsUsed(s, v)
                }

            case *IRReadStatement:
                if s.Dest.IsGlobal() {
                    used[s] = true
                }

                markAsUsed(s, s.Src)

            case *IRWriteStatement:
                markAsUsed(s, s.Src)
                markAsUsed(s, s.Dest)

            case *IRCallStatement:
```

```
                if s.Dest.IsGlobal() {
                    used[s] = true
                }

                for _, argVar := range s.Vars {
                    markAsUsed(s, argVar)
                }

            case *IRSystemCallStatement:
                markAsUsed(s, s.Var)

            case *IRReturnStatement:
                markAsUsed(s, s.Var)

            case *IRIfStatement:
                markAsUsed(s, s.Var)
            }

            return statement
        })

        transformed := Traverse(f, func(statement IRStatement) IRStatement {
            switch statement.(type) {
            case *IRAssignmentStatement, *IRReadStatement:
                if !used[statement] {
                    changed = true
                    return nil
                }
            }

            return statement
        })

        f = transformed.(*IRFunctionDefinition)
    }

    return removeUnusedVariableDeclaration(f)
}
```

3.3 最適化の効果

最適化の効果を例を用いて説明する.

```
// demo/optimize_constant.sc
int main() {
  int a, b;
  int c;
  c = 3;

  a = c; // 3
  b = a + c; // 3 + 3
  print(a + b == 9);  // 3 + 6 == 9
}
```

比較用に最適化を無効化するオプションをつけてコードを生成する.

```
$ ./small-c -optimize=false demo/optimize_constant.sc > demo/optimize_constant.s
$ ./small-c demo/optimize_constant.sc > demo/optimize_constant_optimized.s
```

```
# 最適化前
$ spim -show_stats  -f demo/optimize_constant.s
Loaded: /usr/local/share/spim/exceptions.s
1
--- Summary ---
# of executed instructions
- Total:    47
- Memory:   21
- Others:   26

--- Details ---
       add         2
      addi         9
     addiu         2
      addu         1
       beq         1
       jal         1
        jr         1
        lw        12
       ori         5
       sll         2
        sw         9
   syscall         2
```

```
# 最適化後
$ spim -show_stats  -f demo/optimize_constant_optimized.s
Loaded: /usr/local/share/spim/exceptions.s
1
```

```
--- Summary ---
# of executed instructions
- Total:    22
- Memory:    7
- Others:   15

--- Details ---
      addi         3
     addiu         2
      addu         1
       jal         1
        jr         1
        lw         4
       ori         3
       sll         2
        sw         3
   syscall         2
```

合計命令数が 47 から 22 まで削減された. このように定数畳み込みと無駄な命令を除去を組み合わせると大きな効果がある場合がある.

ただし, 今回の簡単な最適化では, ポインタが多く用いられるようなプログラムではそれほど効果は得られない.

4 課題 13: 相対番地の計算

相対番地の計算は, compile.go の CalculateOffset() 関数に実装している.
複合文を再帰的に探して, それに含まれる変数宣言に対して相対番地を計算している.
グローバル変数の場合は, グローバルポインタ $\$gp$ からの相対番地を計算する.

```
func CalculateOffset(ir *IRProgram) {
  globalOffset := 0
  // global vars
  for _, d := range ir.Declarations {
    size := d.Var.Type.ByteSize()
    globalOffset -= size
    d.Var.Offset = globalOffset
  }

  for _, f := range ir.Functions {
    calculateOffsetFunction(f)
  }
```

```
}

func calculateOffsetFunction(ir *IRFunctionDefinition) {
  offset := 0

  for i := len(ir.Parameters) - 1; i >= 0; i-- {
    p := ir.Parameters[i]
    size := p.Var.Type.ByteSize()

    // arg 4 => 4($fp), arg 5 => 8($fp)
    if i >= 4 {
      p.Var.Offset = (i - 3) * size
    } else {
      p.Var.Offset = offset - (size - 4)
      offset -= size
    }
  }

  minOffset := calculateOffsetStatement(ir.Body, offset)
  ir.VarSize = -minOffset
}

func calculateOffsetStatement(statement IRStatement, base int) int {
  offset := base
  minOffset := 0

  switch s := statement.(type) {
  case *IRCompoundStatement:
    for _, d := range s.Declarations {
      size := d.Var.Type.ByteSize()
      d.Var.Offset = offset - (size - 4)
      offset -= size
    }

    minOffset = offset
    for _, s := range s.Statements {
      statementOffset := calculateOffsetStatement(s, offset)

      if statementOffset < minOffset {
        minOffset = statementOffset
      }
    }
  }
```

```
  return minOffset
}
```

5 課題 15: コード生成

コード生成は, compile.go の Compile() 関数に実装している.
Compile() は中間表現プログラムを受け取り, 生成した MIPS コードを文字列として返す.

変数参照では, グローバル変数の場合は *gp*, それ以外の場合はfp をベースポインタとして計算した相対番地を使って相対参照するようにしている.

```
// Compile takes ir program as input and returns mips code
func Compile(program *IRProgram) string {
  CalculateOffset(program)

  code := ""
  code += ".data\n"
  code += ".text\n.globl main\n"
  for _, f := range program.Functions {
    code += "\n" + strings.Join(compileFunction(f), "\n") + "\n"
  }

  return code
}

func compileFunction(function *IRFunctionDefinition) []string {
  size := function.VarSize + 4*2 // arguments + local vars + $ra + $fp

  var code []string
  code = append(
    code,
    fmt.Sprintf("%s:", function.Var.Name),
    fmt.Sprintf("addi $sp, $sp, %d", -size),
    "sw $ra, 4($sp)",
    "sw $fp, 0($sp)",
    fmt.Sprintf("addi $fp, $sp, %d", size-4),
  )

  for i := len(function.Parameters) - 1; i >= 0; i-- {
    p := function.Parameters[i]
    // arg 4,5,6... is passed via 4($fp), 8($fp), ...
    if i < 4 {
```

```
      code = append(code, fmt.Sprintf("sw $a%d, %d($fp)", i, p.Var.Offset))
    }
  }

  code = append(code, compileStatement(function.Body, function)...)

  code = append(
    code,
    function.Var.Name+"_exit:",
    "lw $fp, 0($sp)",
    "lw $ra, 4($sp)",
    fmt.Sprintf("addi $sp, $sp, %d", size),
    "jr $ra",
  )

  return code
}

func compileStatement(statement IRStatement, function *IRFunctionDefinition) []string {
  var code []string

  switch s := statement.(type) {
  case *IRCompoundStatement:
    for _, statement := range s.Statements {
      code = append(code, compileStatement(statement, function)...)
    }

  case *IRAssignmentStatement:
    code = append(code, assignExpression("$t0", s.Expression)...)
    code = append(code, sw("$t0", s.Var))

  case *IRCallStatement:
    for i := len(s.Vars) - 1; i >= 0; i-- {
      v := s.Vars[i]

      if i >= 4 {
        code = append(code, lw("$t0", v))
        code = append(code,
          "addi $sp, $sp, -4",
          fmt.Sprintf("sw %s, 0($sp)", "$t0"),
        )
      } else {
        code = append(code, lw(fmt.Sprintf("$a%d", i), v))
```

```
    }
  }

  code = append(code, fmt.Sprintf("jal %s", s.Func.Name))
  if len(s.Vars) > 4 {
    code = append(code, fmt.Sprintf("addi $sp, $sp, %d", 4*(len(s.Vars)-4)))
  }
  code = append(code, sw("$v0", s.Dest))

case *IRReturnStatement:
  if s.Var != nil {
    code = append(code,
      lw("$v0", s.Var),
    )
  }

  code = append(code,
    fmt.Sprintf("j %s_exit", function.Var.Name),
  )

case *IRWriteStatement:
  return []string{
    lw("$t0", s.Src),
    lw("$t1", s.Dest),
    "sw $t0, 0($t1)",
  }

case *IRReadStatement:
  return []string{
    lw("$t0", s.Src),
    "lw $t1, 0($t0)",
    sw("$t1", s.Dest),
  }

case *IRLabelStatement:
  return append(code, s.Name+":")

case *IRIfStatement:
  falseLabel := label("ir_if_false")
  endLabel := label("ir_if_end")

  code = append(code,
    lw("$t0", s.Var),
```

```
      fmt.Sprintf("beq $t0, $zero, %s", falseLabel),
    )

    if len(s.TrueLabel) > 0 {
      code = append(code,
        fmt.Sprintf("j %s", s.TrueLabel),
      )
    } else {
      code = append(code,
        fmt.Sprintf("j %s", endLabel),
      )
    }

    code = append(code,
      falseLabel+":",
    )

    if len(s.FalseLabel) > 0 {
      code = append(code,
        fmt.Sprintf("j %s", s.FalseLabel),
      )
    }

    code = append(code,
      endLabel+":",
    )

  case *IRGotoStatement:
    code = append(code, jmp(s.Label))

  case *IRSystemCallStatement:
    switch s.Name {
    case "print":
      return []string{
        "li $v0, 1",
        lw("$a0", s.Var),
        "syscall",
      }
    case "putchar":
      return []string{
        "li $v0, 11",
        lw("$a0", s.Var),
        "syscall",
```

```
      }

    default:
      panic("invalid system call: " + s.Name)

    }
  }

  return code
}

func assignExpression(register string, expression IRExpression) []string {
  var code []string

  switch e := expression.(type) {
  case *IRNumberExpression:
    code = append(code, fmt.Sprintf("li %s, %d", register, e.Value))

  case *IRBinaryExpression:
    leftRegister := "$t1"
    rightRegister := "$t2"

    code = append(code, assignExpression(leftRegister, e.Left)...)
    code = append(code,
      "addi $sp, $sp, -4",
      fmt.Sprintf("sw %s, 0($sp)", leftRegister),
    )
    code = append(code, assignExpression(rightRegister, e.Right)...)
    code = append(code,
      fmt.Sprintf("lw %s, 0($sp)", leftRegister),
      "addi $sp, $sp, 4",
    )

    operation := assignBinaryOperation(register, e.Operator, leftRegister, rightRegister)

    return append(code, operation...)

  case *IRVariableExpression:
    // *(a + 4)
    _, isArrayType := e.Var.Type.(ArrayType)
    if isArrayType {
      return []string{
        fmt.Sprintf("addi %s, %s, %d", register, e.Var.AddressPointer(), e.Var.Offset),
```

```
      }
    }

    return append(code, lw(register, e.Var))

  case *IRAddressExpression:
    return []string{
      fmt.Sprintf("addi %s, %s, %d", register, e.Var.AddressPointer(), e.Var.Offset),
    }
  }

  return code
}

func assignBinaryOperation(register string, operator string, left string, right string) []string
  inst := operatorToInst[operator]
  if len(inst) > 0 {
    return []string{
      fmt.Sprintf("%s %s, %s, %s", inst, register, left, right),
    }
  }

  switch operator {
  case "==":
    falseLabel := label("beq_true")
    endLabel := label("beq_end")

    return []string{
      fmt.Sprintf("beq $t1, $t2, %s", falseLabel),
      li(register, 0),
      fmt.Sprintf("j %s", endLabel),
      falseLabel + ":",
      li(register, 1),
      endLabel + ":",
    }

  case "!=":
    falseLabel := label("beq_true")
    endLabel := label("beq_end")

    return []string{
      fmt.Sprintf("beq $t1, $t2, %s", falseLabel),
      li(register, 1),
```

```
      fmt.Sprintf("j %s", endLabel),
      falseLabel + ":",
      li(register, 0),
      endLabel + ":",
    }

  case ">":
    // a > b <=> (a <= b) < 1
    return append(assignBinaryOperation(register, "<=", left, right),
      fmt.Sprintf("slti %s, %s, 1", register, register),
    )

  case "<=":
    // a <= b <=> a - 1 < b
    return []string{
      fmt.Sprintf("addi %s, %s, -1", left, left),
      fmt.Sprintf("slt %s, %s, %s", register, left, right),
    }

  case ">=":
    // a >= b <=> b <= a
    return assignBinaryOperation(register, "<=", right, left)
  }

  panic("unimplemented operator: " + operator)
}
```

6 感想

最適化の実装は難しかったが, コンパイラの書籍 (ドラゴンブック) を読んだりしながら苦労して実装する中で, コンパイラの奥深さを垣間見ることができた.

今回 Go 言語を使って実装していて思ったのは, Go 言語は表現力に乏しく, 書いていて楽しくない, ということである. モダンなシンタックス, 型による支援, 強力なエコシステムは大きな魅力ではあるが, 他の関数型言語に比べると多くの行を書かなければならない. 次に言語を選ぶときは, 大企業のマーケティングに騙されず, 書いていて楽しい言語を選びたい.