Implementing Register Allocation

Eelco Visser

CS4200 | Compiler Construction | December 2, 2021

Liveness

```
signature
  constructors
    Live : List(RArg) → Anno
rules

  // liveness analysis for straight-line code; no jumps

  uncover-live               :: RProgram → RProgram
  uncover-live-instrs        :: List(RLine) → (List(RArg) * List(RLine))
  live-before(|List(RArg)) :: RLine → (List(RArg) * RLine)

  uncover-live =
    RProgram(uncover-live-instrs; Snd)

  uncover-live-instrs :
    [instr] → (before, [instr'])
    with <live-before(|[])> instr ⇒ (before, instr')

  uncover-live-instrs :
    [instr | instrs] → (before, [instr' | instrs'])
    with <uncover-live-instrs> instrs ⇒ (after, instrs')
    with <live-before(|after)> instr ⇒ (before, instr')

  live-before(|after) :
    instr → (before, instr{Live(after)})
    with <read-write> instr ⇒ (r, w)
    with <union>(<diff>(after, w), r) ⇒ before
```

```
rules

  read-write :: RLine → (List(RArg) * List(RArg))
  vars       :: List(RArg) → List(RArg)

  read-write :
    RLocal(r, _) → ([], [r])

  read-write :
    op#([x | xs]) → (<vars>xs, <vars>[x])
    where <is-operator> op

  read-write :
    _ → ([], [])

  vars = filter(?RVar(_) <+ ?RReg(_))

  is-operator =
    ?"RLi" <+ ?"RAdd"
    <+ ?"RAddi" <+ ?"RMv" <+ ?"RNeg" <+ ?"RMul"
```

Priority Queue

```
signature
  sorts PrioQ
  constructors
    NilQ  : PrioQ
    ConsQ : RArg * int * List(RArg) * PrioQ → PrioQ

rules

  pq-empty :: ? → PrioQ

  pq-empty = !NilQ() // :: PrioQ

  pq-is-empty = ?NilQ()

  pq-next :: PrioQ → RArg

  pq-next : ConsQ(x, _, _, _) → x

rules

  pq-insert(|RArg, RArg)           :: PrioQ → PrioQ
  pq-insert-lst(|RArg, List(RArg))  :: PrioQ → PrioQ

  pq-insert(|x, y) :
    q → <pq-insert(|x, 1, [y])> q

  pq-insert-lst(|x, ys) :
    q → <pq-insert(|x, <length> ys, ys)> q
```

```
rules pq-insert(|RArg, int, List(RArg)) :: PrioQ → PrioQ

  pq-insert(|x, n, ys) :
    NilQ() → ConsQ(x, n, ys, NilQ())
    with debug(!"pq-insert/NilQ: ")

  pq-insert(|y, n, ys) :
    ConsQ(x, m, xs, q) → ConsQ(x, k, zs, q)
    where <eq>(x, y)
    with debug(!"pq-insert/ConsQ/1: ")
    with <union>(xs, ys) ⇒ zs // :: List(a)
    with <length>zs ⇒ k :: int
    with <debug(!" ⇒ ")>ConsQ(x, k, zs, q)

  pq-insert(|x, n, ys) :
    ConsQ(z, i, zs, q) → <pq-sort(|z, i, zs)> q'
    where <not(eq)> (x, z)
    with debug(!"pq-insert/ConsQ/2: ")
    with <pq-insert(|x, n, ys)> q ⇒ q'

rules pq-sort(|RArg, int, List(RArg)) :: PrioQ → PrioQ

  pq-sort(|x, n, xs) :
    ConsQ(y, m, ys, q) → ConsQ(y, m, ys, q')
    where <gt>(m, n)
    with <pq-sort(|x, n, xs)> q ⇒ q'

  pq-sort(|x, n, xs) :
    q → ConsQ(x, n, xs, q)
```

Build Interference Graph

```
rules

  build-interference-graph :: RProgram → PrioQ

  build-interference-graph =
    ?RProgram(<foldr(pq-empty, make-edges1)>)

  make-edges1 :: RLine * PrioQ → PrioQ

  make-edges1 :
    (instr{Live(l)}, q1) → q3
    with <read-write> instr ⇒ (_, w)
    with <filter(where(<not(member)>(<id>, w)))> l ⇒ l'
    with <foldr(!q1, \(x, q) → <pq-insert(|x, <length>l', l')>q\)> w ⇒ q2
    with <foldr(!q2, \(x, q) → <pq-insert(|x, <length>w, w :: List(RArg))>q\)> l'
```

Allocate Registers (1)

```
allocate-registers :: RProgram → RProgram
color-graph        :: RProgram → List((RArg * int))
color-variables    :: RProgram → RProgram

allocate-registers =
  where(color-graph)
  ; color-variables

color-graph =
  build-interference-graph
  ; pq-color

pq-color         :: PrioQ → List((RArg * int))
color-vertex     :: (RArg * List(RArg)) → (RArg * int)
find-color(|int) :: List(int) → int

pq-color : NilQ() → []

pq-color :
  ConsQ(x, _, xs, q) → [elem | elems]
  with <color-vertex>(x, xs) ⇒ elem
  with <pq-color> q ⇒ elems

color-vertex :
  (x, xs) → (x, color)
  with <filter(Color)> xs ⇒ colors
  with <find-color(|0)> colors ⇒ color
  with rules( Color : x → color )
```

```
rules

  find-color(|n) :
    [] → n
  find-color(|n) :
    [c | cs] → n
    where <gt> (c, n)
  find-color(|n) :
    [n | cs] → c'
    where <find-color(|<inc>n)> cs ⇒ c'
  find-color(|n) :
    [c | cs] → c’
    where <find-color(|<inc>c)> cs ⇒ c'
```

Allocate Registers (2)

```
rules

  color-variable             :: TP // RArg → RArg
  color-to-location(|RArg) :: int → RArg

  color-variables =
    alltd(color-variable) // <+ color-variable)

  color-variable :
    RLocal(v@RVar(x), t) :: RLine → I(RNop(), $[[x]: [<pp-type>t] ⇒ [<int-to-string>color]]) :: RLine
    where <Color> RVar(x) ⇒ color
    with <color-to-location(|v)> color ⇒ loc

  color-variable :
    (v@RVar(x)) :: RArg → loc :: RArg
    where <Color>v ⇒ i
    with <color-to-location(|v)> i ⇒ loc

  color-to-location(|x) :
    i → RReg($[t[<int-to-string>i]])
    where <geq>(i, 0); <lt>(i, 6) // number of temporary registers

  color-to-location(|x) :
    i → RMem(<int-to-string>off, "fp")
    where <gt>(i, 5) // number of temporary registers
    with <var-offset-get
          <+ (var-offset-set(|x, <stack-inc(| -4)>)
              ; <var-offset-get>x)> i ⇒ off
```

Compiling Control Flow

Eelco Visser

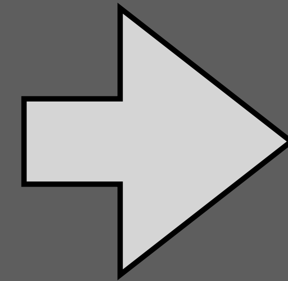
CS4200 | Compiler Construction | December 2, 2021

Extend ChocoPy Signature

```
signature
  constructors
    IfExp     : Exp * Exp * Exp → Exp
    Let       : ID * Type * Exp * Exp → Exp
    IfStat    : Exp * Block * Block → Statement
    BlockStat : Block → Statement
```

Desugar

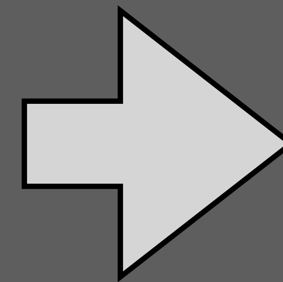
```
a : bool = False
b : bool = True
not(a and b)
```



```
Program(
  [ VarDef(TypedVar("a", Type("bool")), False())
  , VarDef(TypedVar("b", Type("bool")), True())
  ]
, [ Exp(
      IfExp(
        IfExp(Var("a"), Var("b"), False())
      , False()
      , True()
      )
    )
  ]
)
```

Desugar

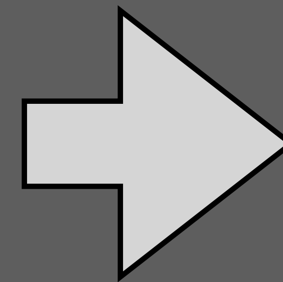
```
a : int = 1
b : int = 2
a ≠ b
```



```
Program(
  [ VarDef(TypedVar("a", Type("int")), Int("1"))
  , VarDef(TypedVar("b", Type("int")), Int("2"))
  ]
, [Exp(
     IfExp(Eq(Var("a"), Var("b")), False(), True())
   )]
)
```

Desugar

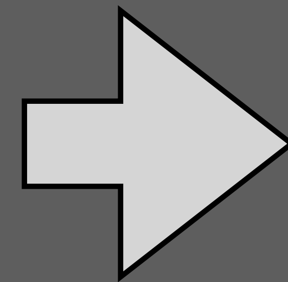
```
a : int = 1
b : int = 2
if a ≠ b:
  3
else:
  4
```



```
Program(
  [ VarDef(TypedVar("a", Type("int")), Int("1"))
  , VarDef(TypedVar("b", Type("int")), Int("2"))
  ]
, [ IfStat(
      Eq(Var("a"), Var("b"))
    , Block([Exp(Int("4"))])
    , Block([Exp(Int("3"))])
    )
  ]
)
```

Desugar

```
x : bool =
False
y : int = 0
z : int = 1
x = (y ≠ z)
```



```
Program(
  [ VarDef(TypedVar("x", Type("bool")), False())
  , VarDef(TypedVar("y", Type("int")), Int("0"))
  , VarDef(TypedVar("z", Type("int")), Int("1"))
  ]
, [ IfStat(
      Eq(Var("y"), Var("z"))
    , Block([Assign([Target(Var("x"))], False())])
    , Block([Assign([Target(Var("x"))], True())])
    )
  ]
)
```

Extend C-IR

```
signature
  constructors
    CLt   : CAtom * CAtom → CExp
    CEq   : CAtom * CAtom → CExp

    CNot  : CAtom → CExp

    CGoto : CLabel → CTail
    CIf   : CExp * CLabel * CLabel → CTail
```

Generating Control Flow Blocks

```
signature
  sorts Future
  constructors
    Promise : string → Future
rules

  block-to-goto(|string) :: CTail → Future
  jump-to-block :: Future → CLabel

  block-to-goto(|schema) :
    tail → Promise(scrt)
    with <newname> schema ⇒ lbl
    with !CBlock(CLabel(lbl), tail) ⇒ block
    with <newname> "secret" ⇒ scrt
    with rules( Future : scrt → block )

  jump-to-block :
    Promise(scrt) → lbl
    with <Future> scrt ⇒ res
    with if !res ⇒ block@CBlock(lbl, _) then
           <add-cfg-node> block
           ; rules( Future : scrt → lbl )
         else
           !res ⇒ lbl
         end
```

Explicate Control (Sketch)

```
rules

  explicate-tail-stm :
    IfStat(e, Block(stats1), Block(stats2)) → tail
    where <explicate-tail-seq; block-to-goto(|"s")> stats1 ⇒ s
    where <explicate-tail-seq; block-to-goto(|"f")> stats2 ⇒ f
    where <explicate-pred(|s, f)> e ⇒ tail

rules

  explicate-pred(|Future, Future) :: Exp → CTail

  explicate-pred(|s, f) :
    Var(x :: string) → CIf(CEq(CVar(x), CInt("0")),
                           <jump-to-block>f,
                           <jump-to-block>s)

  explicate-pred(|s, f) :
    Int(i) → CGoto(lbl)
    with if <eq>(i, "0") then !f else !s end; jump-to-block ⇒ lbl

  explicate-pred(|s, f) :
    Lt(a1, a2) → CIf(CLt(<explicate-atom>a1, <explicate-atom>a2),
                     <jump-to-block>s,
                     <jump-to-block>f)

  explicate-pred(|s, f) :
    Eq(a1, a2) → CIf(CEq(<explicate-atom>a1, <explicate-atom>a2),
                     <jump-to-block>s,
                     <jump-to-block>f)
```

Except where otherwise noted, this work is licensed under