Optimizing compiler

Andrew Lelechenko
1@dxdy.ru

#kievfprog, Kiev, 18.03.2017

JFP **19** (1): 27–45, 2009.
doi:10.1017/S0956796808007016 First published online 1 October 2008 Printed in the United Kingdom

Commercial uses: Going functional on exotic trades

SIMON FRANKAU

Barclays Capital, 5 The North Colonnade, London E14 4BB, UK
(*e-mail:* `Simon.Frankau@barclayscapital.com`)

DIOMIDIS SPINELLIS

Athens University of Economics and Business, Patision 76, GR 104 34, Athens, Greece
(*e-mail:* `dds@aueb.gr`)

NICK NASSUPHIS and CHRISTOPH BURGARD

Barclays Capital, 5 The North Colonnade, London E14 4BB, UK
(*e-mail:* `{Nick.Nassuphis,Christoph.Burgard}@barclayscapital.com`)

Papers and slides

- Simon Frankau, Diomidis Spinellis, Nick Nassuphis, Christoph Burgard. *Commercial Uses: Going Functional on Exotic Trades.* J. Funct. Program. 19, 1, 27–45.
- Tim Williams. *Exotic Tools for Exotic Trades* @ CodeMesh 2013.
- Tim Williams, Peter Marks. *Structural Typing for Structured Products* @ Haskell Exchange 2014.

Why not Haskell?

- Haskell is an amazing language for writing your own compiler.
- A rich ecosystem of libraries dedicated to compiler-related tasks.
- A really powerful type system that can eliminate large classes of errors at compile time.
- Many excellent educational resources for compiler writers:
    - Stephen Diehl. *Write You a Haskell.*
    - Andres Löh, Conor McBride, Wouter Swierstra. *A Tutorial Implementation of a Dependently Typed Lambda Calculus.*
- Compilers, written in Haskell: GHC, Elm, Purescript, Idris, Agda...

SIMPL — Simple IMPerative Language

- Scalar and vector `Double`-valued variables: `a`, `b[i]`, `c[i][j]`.
- Vector length is statically known beforehand.
- Arithmetic operators, comparisons and ternary operator.
- Only `for`-loops without preliminary `break`.
- Basically, its computational power is equivalent to Excel sheet.
- Powerful enough for physical/financial Monte-Carlo simulations with calculations repeated over random inputs and step-by-step time.

Why not Fortran?

Example: *n*-body problem, youtube.com/embed/qIVe_xEv6zQ

- m[i] is the mass of i-th object (is known in compile time),
- x[i] and y[i] are its coordinates,
- vx[i] and vy[i] are its velocities by X- and Y-axis,
- ax[i] and ay[i] are its accelerations.

```
dt = 0.001
G  = 6.67e-11
foreach i in 1..n
  ax[i] = 0
  ay[i] = 0
  foreach j in 1..n
    d[i][j] = (x[i] - x[j]) ^ 2 + (y[i] - y[j]) ^ 2
    ax[i] = ax[i] - G * m[j] * (x[i]-x[j]) / d[i][j] ^ 3/2
    ay[i] = ay[i] - G * m[j] * (y[i]-y[j]) / d[i][j] ^ 3/2
foreach i in 1..n
  vx[i] = vx[i] + ax[i] * dt
  vy[i] = vy[i] + ay[i] * dt
   x[i] =  x[i] + vx[i] * dt
   y[i] =  y[i] + vy[i] * dt
```

Abstract syntax tree: types

Loop counters and variables are annotated with sizes on type level.

```haskell
data Counter (range :: Nat) = Counter String
data Reference (arity :: [Nat]) where
  V    :: String -> Reference xs
  (:!) :: Reference (x : xs) -> Counter x -> Reference xs
```

Expressions are simple:

```haskell
data Expr a where
  Ref  :: Reference '[] -> Expr Double
  Num  :: Double -> Expr Double
  (:+) :: Expr Double -> Expr Double -> Expr Double
  (:*) :: Expr Double -> Expr Double -> Expr Double
  (:^) :: Expr Double -> Expr Double -> Expr Double
  (:<) :: Expr Double -> Expr Double -> Expr Bool
  If   :: Expr Bool -> Expr a -> Expr a -> Expr a
```

And statements are even simpler:

```haskell
data Stmt where
  (:=) :: Reference '[] -> Expr Double -> Stmt
  For  :: Counter x -> [Stmt] -> Stmt
```

Abstract syntax tree: utils

```haskell
instance Num (Expr Double) where
  (+) = (:+)
  (*) = (:*)
  negate x = x :* (-1)
  abs x    = If (x :< 0) (negate x) x
  signum x = If (x :< 0) (-1) (If (0 :< x) 1 0)
  fromInteger = Num . fromInteger

(&&) :: Expr Bool -> Expr Bool -> Expr Bool
a && b = If a b a

(||) :: Expr Bool -> Expr Bool -> Expr Bool
a || b = If a a b
```

Code example

For a given c build a vector `[c, c+1...]`, square up its elements and return their sum.

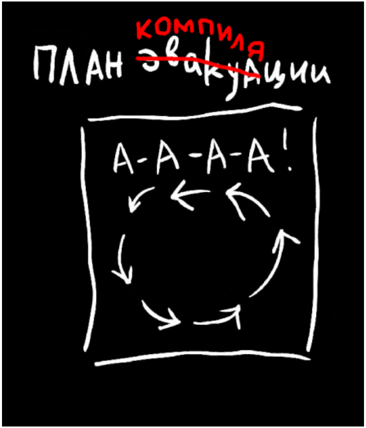
```
[ For i
  [ a!i := Ref c
  , c   := Ref c + 1 ]
, For i
  [ b!i := Ref (a!i) :^ 2 ]
, ret   := 0
, For i
  [ ret := Ref ret + Ref (b!i) ] ]
where a = "a"; b = "b"; c = "c"; ret = "ret"; i = "i"
```

Optimized:

```
[ ret   := 0
  For i
  [ ret := Ref ret + Ref c :^ 2
  , c   := Ref c + 1 ]
where c = "c"; ret = "ret"; i = "i"
```

Compiler's architecture

Lexer/Parser	→	Abstract Syntax Tree	→	Code Generator
`parsec`,	→	optimizations	→	ASM, C--, LLVM,
`alex` / `happy`	→		→	C, JavaScript



Marginalia: lexer and parser

Semantically:

- Lexer reads a list of tokens.
- Parser transforms tokens into syntax tree.

Grammars:

- Lexer reads a regular grammar (regexp without backtracking).
- Parser reads a context-free grammar (can parse nested brackets).
- parsec can read even a context-sensitive grammar.

Static advantages:

- Lexer generates a very efficient finite automaton.
- Parser checks grammar for ambiguities.

Gentle optimizations

We are interested in optimizations of SIMPL such that

- no auxiliary statements or variables are introduced,
- expressions are replaced with equivalent in runtime ones.

We cannot:

- dismantle expressions into three-address code:
  `x = y * 3 + 2` $\not\Longrightarrow$ `t = y * 3; x = t + 2`.
- dismantle loops into blocks, labels and gotos;
- use Single Static Assignment or Continuation Passing Style.

We are free to

- rewrite expressions;
- eliminate dead statements;
- shuffle statements both top-down and inside-out;
- detect high-level patterns such as maps and folds.

Local optimizations

Despite SIMPL is an imperative language with global side effects, its expressions are pure and has no effects.

- Constant folding:
  `2 :* 3` $\Longrightarrow$ 6.
- Short circuiting:
  `If (0 :< 1) a b` $\Longrightarrow$ `b`.
- Algebraic simplifications:
  `x :+ 0` $\Longrightarrow$ `x`, `x :* 0` $\Longrightarrow$ 0.
- Reassociation and redistribution:
  `(x :+ 2) :+ 3` $\Longrightarrow$ `x :+ (2 :+ 3)` $\Longrightarrow$ `x :+ 5`,
  `(x :+ 3) :* 2` $\Longrightarrow$ `x :* 2 :+ 3 :* 2` $\Longrightarrow$ `x :* 2 :+ 6`.

Dead code: a time to keep, and a time to cast away

- Detect statements, which can be entirely removed.
- The analysis goes *backwards* in terms of control flow, from the return statement up to the start.
- Transform as you go: if left-hand side reference is not *live set*, remove the statement at once. Otherwise remove LHS reference from *live set* and add RHS references.

```
          -- Live set
a = 1     -- {}
b = a + 2 -- {a}, remove
c = a + 3 -- {a}
d = b * 2 -- {c}, remove
ret = c   -- {c}
          -- {ret}
```

Dead code: loops

- Loops change control flow and are executed one or more times.
- Perform a dummy backward pass without striping of statements to collect circular references.
- Then unite obtained live set with the live set at the end of loop and perform actual pass.

```
a = 0          -- {}
for i
  b[i] = a     -- {a} / {a}
  a = b[i] + 1 -- {b} / {b}
end            -- {b} / {a, b}
r = 0          -- {b}
for i
  r = r + b[i] -- {r, b} / {r, b}
end            -- {r}   / {r, b}
```

```
a = 0          -- {}, remove
for i
  b[i] = a     -- {} / {}, remove
  a = b[i] + 1 -- {} / {}, remove
end            -- {} / {}
r = 0          -- {}
               -- {r}
```

Inlining: a time to rend, and a time to sew

- Substitute RHS expression instead of LHS reference.
- The analysis goes *forward* in terms of control flow.
- Transform as you go: use the result of inlining for further substitutions.

Use cases:

- Always inline, when RHS is a constant or a reference.
- Always inline live variables, which are consumed only once.

```haskell
data Liveness = One | Many
instance Semigroup Liveness where
  _ <> _ = Many
```

- Inline if it gives way to non-growing expression after local optimizations.

Inlining: example

```
a = 2
b = x + a
c = b
d = c + 1
e = c * 2
f = c ^ 3
g = d * e * f * f
```

```
a = 2
b = x + 2 -- inline constants
c = x + 2 -- b is a one-shot variable
d = x + 3 -- non-growing local optimizations
e = c * 2 -- x * 2 + 4 is larger
f = c ^ 3 -- (x + 2) ^ 3 is larger
g = (x + 3) * (c * 2) * f * f
          -- d and e are one-shot variables
```

Loop fusion: a time to break down and a time to build up

- Detect maps and folds in imperative code. Loops, which effect is invariant to permutation of iterations' order, are `map`. Other loops are `fold`.
- Fuse using `map`/`map` and `map`/`fold` rules.

```
for i
  y[i] = x[i] * 2
for i
  z[i] = y[i] * 3
```

$$\text{map f . map g} = \text{map (f . g)}$$

```
for i
  y[i] = x[i] * 2
  z[i] = y[i] * 3
```

Loop fusion: map/fold rule

```
for i
  y[i] = x[i] * 2
ret = 0
for i
  ret = ret + y[i] * 3
```

Shuffle blocks:

```
ret = 0
for i
  y[i] = x[i] * 2
for i
  ret = ret + y[i] * 3
```

$$\text{foldr f x . map g} = \text{foldr (f . g) x}$$

```
ret = 0
for i
  y[i] = x[i] * 2
  ret = ret + y[i] * 3
```

Loop fission - 1

```
ret = 0
for i
  y[i] = x[i] * 2
  ret = ret * 2 + y[i]
for i
  z[i] = y[i] * 3
```

Distribute first loop:

```
ret = 0
for i y[i] = x[i] * 2
for i ret = ret * 2 + y[i]
for i z[i] = y[i] * 3
```

Shuffle blocks:

```
ret = 0
for i y[i] = x[i] * 2
for i z[i] = y[i] * 3
for i ret = ret * 2 + y[i]
```

Loop fission - 2

```
ret = 0
for i y[i] = x[i] * 2
for i z[i] = y[i] * 3
for i ret = ret * 2 + y[i]
```

Apply map/map rule:

```
ret = 0
for i
  y[i] = x[i] * 2
  z[i] = y[i] * 3
for i
  ret = ret * 2 + y[i]
```

Apply map/fold rule:

```
ret = 0
for i
  y[i] = x[i] * 2
  z[i] = y[i] * 3
  ret = ret * 2 + y[i]
```

Thank you!