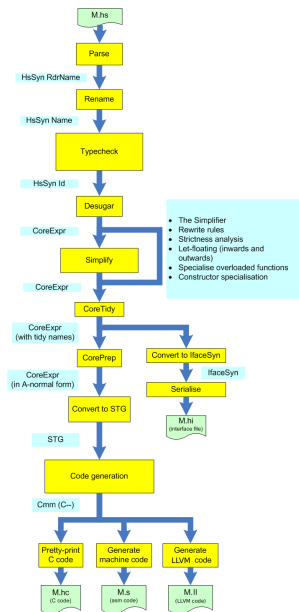


# GHC へのプログラム変換パスの追加

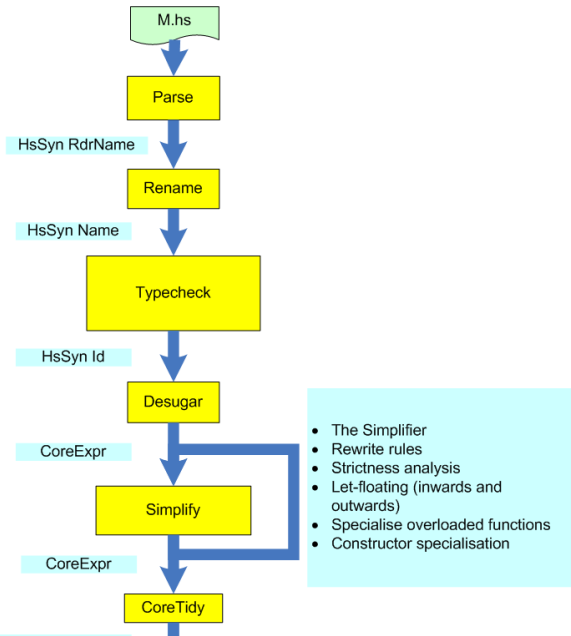
日比野 啓

September 30, 2012

# 今日の話



# 今日の話



# Haskell から Core へ

```
module Foo (f) where

f :: Int -> Int -> Int
f x y = x * (x + y)
```

---

Haskell を Core 形式へ変換し、出力するコマンド  
% ghc -c -ddump-simpl Foo.hs

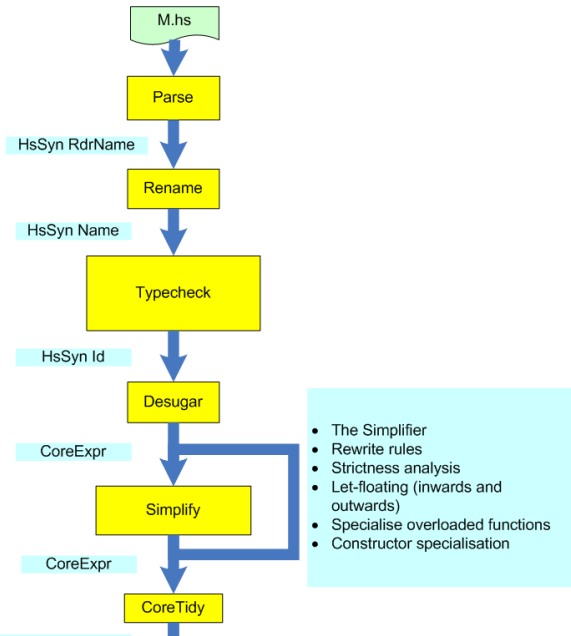
# Haskell から Core へ

```
Foo.f :: GHC.Types.Int -> GHC.Types.Int ->
      GHC.Types.Int
[GblId, Arity=2]
Foo.f =
  \ (x_a9H :: GHC.Types.Int)
    (y_a9I :: GHC.Types.Int) ->
    GHC.Num.*
      @ GHC.Types.Int
      GHC.Num.$fNumInt
      x_a9H
      (GHC.Num.+
        @ GHC.Types.Int GHC.Num.$fNumInt
        x_a9H y_a9I)
```

---

Core は GHC が内部で利用している Lambda 式

# CoreからCoreへ



# Core から Core へ

Lambda 式を変換すれば、GHC でプログラム変換が書ける。主に最適化のパスを実装する目的で利用されているらしい

- compiler/simplCore 以下に変換パスの定義
- compiler/coreSyn 以下に Core の定義や変換用ライブラリ

# simpl の変換パスの追加

例えば foo というパスを追加する

---



# simpl の変換パスの追加

## 変換パス foo の追加

---

compiler/simplCore/Foo.lhs:

```
module Foo (fooProgram) where
```

```
import CoreSyn (CoreProgram)
```

```
fooProgram :: CoreProgram -> CoreProgram  
...
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

compiler/simplCore/Foo.hs:

```
module Foo (fooProgram) where
```

```
import CoreSyn (CoreProgram)
```

```
fooProgram :: CoreProgram -> CoreProgram  
fooProgram = id
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

```
compiler/simplCore/CoreMonad.lhs
```

```
data CoreToDo ...
```

```
...
```

```
| CoreFoo
```

```
...
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

compiler/man/DynFlags.hs

```
data DynFlag
    ...
    -- optimisation opts
    ...
    | Opt_Foo
    ...
fFlags = [
    ...
    ( "foo",  Opt_Foo ,  nop ),
    ... ]
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

compiler/coreSyn/SimpleCore.lhs

```
...
getCoreToDo
  ...
  where
    ...
    strictness = dopt Opt_Strictness dflags
    ...
    foo        = dopt Opt_foo dflags
    ...
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

```
compiler/coreSyn/SimpleCore.lhs
```

```
...
getCoreToDo
  ...
  [ ...
    runWhen strictness ... ,
    ...
    runWhen foo CoreFoo ,
    ...
  ]
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

```
compiler/coreSyn/SimpleCore.lhs
import Foo (fooProgram)
...
doCorePass :: CoreToDo -> ...
...
doCorePass CoreFoo = doPass fooProgram
...
```

# simpl の変換パスの追加

## 変換パス foo の追加

---

- ここまでの変更で、コンパイラに `-O1 -ffoo` スイッチを与えれば、`Foo (fooProgram)` でプログラムが変換されるようになっているはず。
- あとは `Foo.hs` を実装すればよい



# 利用できるライブラリ

外部ライブラリと同じものでは

- ex. `-package Cabal-1.14.0 -package array-0.4.0.0 -package base-4.5.0.0 -package bin-package-db-0.0.0.0 -package bytestring-0.9.2.1 -package containers-0.4.2.1 -package directory-1.1.0.2 -package filepath-1.3.0.0 -package hoopl-3.8.7.3 -package hpc-0.5.1.1 -package old-time-1.1.0.0 -package process-1.1.0.1 -package unix-2.5.1.0`

# 利用できるライブラリ

## コンパイラ内部だと

- `TrieMap(CoreMap)`
  - `CoreExpr` を key とした Map
- `CoreSubst(Subst)`
  - `Core Syntax` を置換するためのデータ構造

# 実装されている最適化パスの例

CSE (Common SubExpression Elimination)

- -O1 -fcse
- PRE (Partial redundancy elimination) とも
- 式の字面で判断して共通式を除去

```

x = p + q
y = p + q
--
x = p + q      -- Map  p + q => x
y = p + q      -- 置き換え対象
--
x = p + q      -- Map  p + q => x
y = x

```

```
r = p
s = q
x = p + q
y = r + s
--
r = p
s = q
x = p + q      -- Map  p + q => x
y = r + s      -- Map  r + s => y
-- 置き換え対象が無い
```

# 自分でも最適化パスを追加してみました

GVN (Global value numbering)

- 式が生成する値ごとに異なる番号を付けていき、同じ番号の付いた式を除去する

# GVN

--  $p, q$  は自由変数

$r = p$

$s = q$

$x = p + q$

$y = r + s$

$z = x * y$

# GVN

```
z = x * y  -- 略
-- hash[r + s] = hplus[hash[r],
--                hash[+],
--                hash[s]]
--                -- p => 1, (+) => 2, q => 3,
--                -- r => 1, s => 3
--                = hplus[ 1, 2, 3]
--                = 123
-- この例では例えば 123 になったとしておく
y = r + s  -- y => 123
x = p + q  -- x => 123
s = q      -- s => 2, q => 2
r = p      -- r => 1, r => 1
```



# GVN

```
y = r + s  -- y => 123
x = p + q  -- x => 123
s = q      -- s => 2, q => 2
r = p      -- r => 1, r => 1

-- 1:      r ==> p
-- 2:      s ==> q
-- 123:    y ==> x
```

# GVN

```
-- 1:      r ==> p
-- 2:      s ==> q
-- 123:    y ==> x
```

```
r = p          -- 変化なし          -- 除去可能
s = q          -- 変化なし          -- 除去可能
x = p + q      -- 変化なし
y = p + q      -- <-- y = r + s    -- 除去可能
z = x * x      -- <-- z = x * y
```

デモ

デモ

# まとめ

- Lambda 式 Core の変換でプログラム変換を実装できる
- ライブラリも結構使えるので書きやすい
- GVN を実装してみた。が不完全なのでもうちょっとがんばりたい
- Compiler plugin 化もしてみたい