2015 年 5 月

2015-05-27 (Wed)

おなかいたいので、簡単な確認のみ。

↓こんな風に、型宣言と関数定義が離れていても OK

```
f :: [Int] -> Int
g :: [Int] -> Int
main :: IO ()

f [] = 0
f (x:xs) = x + f xs

g [] = 1
g (x:xs) = x * g xs

main = do putStrLn $ show $ f [1..10]
          putStrLn $ show $ g [1..10]
```

↓これでもいい。

```
f [] = 0
f (x:xs) = x + f xs

g [] = 1
g (x:xs) = x * g xs

main = do putStrLn $ show $ f [1..10]
          putStrLn $ show $ g [1..10]

f :: [Int] -> Int
g :: [Int] -> Int
main :: IO ()
```

↓こういうのはだめなんだけど

```
f [] = 0
g [] = 1
f (x:xs) = x + f xs
g (x:xs) = x * g xs
```

というわけで、renIDecls では、renDecls ds' の手前でメソッドに対する型宣言を作成して、renDecls (ds"++ds') なり renDecls (ds'++ds") なりとしてやればよいだろう。

renDecls による型宣言 (TypeSigDecl) から Scheme の変換では quantify されないので、toBg のなかでやる。

toBg は 2 pass で、1 pass 目では型宣言は dictionary に蓄積しつつ、関数定義をまとめる。そして、2 pass 目で型宣言 dictionary をつかって、Impl/Expl に変換。

本当はそこで SCC いるけど、それは後回しにする予定。(SCC しないとどうなるか試して、理解してからでもよいだろう)

2015-05-26 (Tue)

[Bunny] toBg の前に

昨日 Renamer おわりと書いたが、型推論の前に toBg が必要で、さらにその前に、インスタンス宣言のメソッド定義に対して型シグネチャを生成してやる必要がある(と思う)。

たとえば、現状の出力では、

```
("Main.return",Just (Forall [Kfun Star Star,Star] ([IsIn "Main.Monad" (TGen 0)] :=> TAp (TAp (TCon (
```

```
("Main.%IO.return",Nothing,[([],Const ("Prim.primRetIO" :>: Forall [Star] ([] :=> TAp (TAp (TCon (Ty
```

のようになっていて、Main.return には型宣言のみがあって関数定義がなく、Main.%IO.return には型宣言がなくて、関数定義だけがある。これに対して、Main.return の型宣言に型代入をほどこして、Main.%IO.return の型宣言をつくってやらなければならない。

しかし、ここは「型代入」の出番かなと思ったものの、型代入は TGen を無視してしまうぞ。ということは、quantify 前の型宣言を覚えておかないといけないということですね。

renCDecl ではまだ quantify しないことにしてみようか。そうすると、Main.return の tempbind は次のようになる:

```
("Main.return",Just (Forall [] ([IsIn "Main.Monad" (TVar (Tyvar "m" (Kfun Star Star)))] :=> TAp (TAp
```

これなら、

```
Forall [Star] ([] :=> TAp (TAp (TCon (Tycon "(->)" (Kfun Star (Kfun Star Star)))) (TGen 0)) (TAp (TC
```

に変形するのはどうにかできそう。renCDecl, renIDecl を終えて、toBg の中で Expl の Schem を quantify するようにしてやればよさそう。

それはそうと、”(->)” が qualifeid name になってないな。

このへんで、また明日。

2015-05-25 (Mon)

[Bunny] Renamer おわり

ひとまず、tqd.hs + minlibs.hs が処理できる程度の Renamer は作った。次からいよいよ型推論である。

2015-05-23 (Sat)

そういえば、MathJax を導入しているのに、全然つかってなかったので英語版 Math Girls の表紙にもなっているあの数式 (式 1) を書いておいたり。

$$\sum_{k=0}^{\infty} \heartsuit^k \tag{1}$$

昨日の変更で template に施したはずの修正が先祖がえりしてしまってたことに気づいたので、よかった。

Bunny の作成は、Prim.error を PreDefined.hs に足して、Class Decl は通った。次は Instance Decl の処理、その後、いよいよ型推論へ。型推論への入力の段階 (Typing.Expr) の Pretty Print ほしいところだけど…、実際には Core に対してつくるかなぁ。

予定どおり、6 月までに Renamer を終えて型推論にはいれそう。今年中にひとまずの完成をみたいので、前半（6 末まで）にサンプルプログラムの疎通はしておきたいんだよなぁ。あ、「予定」では疎通 8 末って書いてるなぁ。そっか、一ヶ月で Core, STG, CodeGen すべては無理だよな。この 3 つを各一ヶ月でも、それなりにコンスタントにがんばらないと。

むむ。上の「予定」からのリンク、lang の部分がおかしいような。相対リンクにしたほうがよさそうかな。相対リンクにしてみる。

2015-05-21 (Thu)

Haskell と仲良くなりたくて、という話。

要約：コンパイラ書きたいんだけど、コンパイラが書きたいわけじゃなくて、という話。

Haskell コンパイラを書きたいという動機に関するメモ。

Haskell ができるようになりたくて、いくつか本を読んでみたりしたのだけれど、なかなか身につかない感じ。これには、大きくわけて 2 つの理由がある気がしました。

ひとつは、それなりの分量のプログラムを実際に書いてみないことには身につかないだろうという点。これだけであれば、Haskell に限らない話なので特段どうというわけではないのですが。そこで二点め。こちら

が、私が Haskell の場合に特に強く感じていたものということになります。

本を読んで、その都度ためしてみたりすれば、「Haskell でこう書けば、こうなる」といった「理解」はできていくように思えました。ですが、どうしてコンピューターがそのように動くのか、しっくり理解できない。いつまでたってもできない。これは、漠然とした不安なのですが、この調子で Haskell に『慣れて』いっても無駄なのではないかというような不安。

たとえば、Assembly Language でプログラムを書けば、コンピュータはこういうものであるという理解の上で、面倒ではあるけど直接的な方法で、自分がコンピュータの動作を指定しているという安心感がありました。実は、OS やらライブラリやら、または、そのさらに下のハード層をきちんと隅々まで理解しているとは限らず、それは、ある種の偽りの安心感ではあったのですが、とにかくうれしいし、安心。

そして、C で書くときにも、同種の満足感が失われることはなく。C コンパイラは十分優秀でしたが、コンパイラが何をしているのか「さっぱりわからない」わけではない。実際にコンパイラを書ける実力を持っていたわけではないのですが、C で書かれたコードと同等のものを Assembly でも書けるよというのが、なんというか、自信の根拠みたいなものになっていました。

その後、Perl や Ruby をつかうことになって、便利さ、そして、自分にとっての「謎さ」加減は増すのですが、これらの言語は所謂 C 族で、「なにをやっているのか、さっぱりわからん」みたいな不安まではいかない。正直、処理系がどうやってるのかわからないところもあるのですが、まだ小さな不安でした。

ところが、Haskell です：

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

ぐぉぉ、なんだこれは。

時間をかければ、そういう「意味」になってそうなのはわかる。だけど、これは、いままで安心して楽しんできた「プログラミング」とは随分と異なります。なんでこれで「動く」んだろう？！

このままでは、どうにか Haskell でプログラミングできるようになったとしても、かつてのような安心感は得られず、どこか不安なままなのではないか。

これを克服したい。というわけで、例の 2 点をどうにかしたいと考えました。

繰り返すと、私が Haskell とすっきり仲良くなれない理由は以下の二点だといえそうでした：

- 書く分量が足りていない
- コンパイラがなにやっているのかわからなくて、他人の魔法に頼っているみたいで嫌だ

なれば、自分でコンパイラ書いてみれば、ふたつとも一挙に解決するんじゃないか？

というわけ。

[Bunny] Kind Inference

型推論の前に Kind Inference しなければならなかった。

ちょっとやっつけっぽいけど書いていって、いまは

```
  fail   :: String -> m a
```

のところでとまっている。

```
...
(Tycon (Name {orig_name = "String", qual_name = "", name_pos = (20,13), isConName = True}),"kiExpr")
Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"),(":","Prim.:"),("<=","Prim.<="),(">"
```

2015-05-20 (Wed)

[Bunny] Class 宣言

Renamer (と呼ぶと若干名が体を表してないのだが) における Class 宣言の扱いにはいる。
以下の例にて、Renamer がどういう情報を抽出しないといけないかというと、

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  -- Minimal complete definition: (>>=), return
  p >> q = p >>= \_ -> q
  fail s = error s
```

まず、class 定義にあたる EnvTransformer, `addClass "Monad" []` と、各メソッドの型宣言にあたる情報、

```
cMonad = "Monad"

returnMfun
 = "return" :>: (Forall [Kfun Star Star, Star]
                  ([isIn1 cMonad (TGen 0)] :=>
                   (TGen 1 `fn` TAp (TGen 0) (TGen 1))))
mbindMfun
 = ">>=" :>: (Forall [Kfun Star Star, Star, Star]
               ([isIn1 cMonad (TGen 0)] :=>
```

```
                (TAp (TGen 0) (TGen 1) `fn` (TGen 1 `fn` TAp (TGen 0) (TGen 2)) `fn` TAp (TGen 0) (T
mthenMfun
 = ">>" :>: (Forall [Kfun Star Star, Star, Star]
              ([isIn1 cMonad (TGen 0)] :=>
               (TAp (TGen 0) (TGen 1) `fn` TAp (TGen 0) (TGen 2) `fn` TAp (TGen 0) (TGen 2))))
failMfun
 = "fail" :>: (Forall [Kfun Star Star, Star]
                ([isIn1 cMonad (TGen 0)] :=>
                 (tString `fn` TAp (TGen 0) (TGen 1))))
```

さらに、メソッドのデフォルト定義も。これらは、トップレベルの Bind になるんですね。

よし、ここまでめざしてやってみよう。

→今日は、addClass "Main.Monad" 後、ClassDecl 中の decls に対して renDecl するところまで。TypeSigDecl が未実装なので、このまま実装すると (Main.>>=) 等が名前登録されず、後に参照したところでエラーする。

```
...
Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"),(":","Prim.:"),("<=","Prim.<="),(">"
drive_semant: qname not found: >>=
```

つづきは、

- TypeSigDecl 対応、上に述べたように、TypeSigDecl でも qualName の登録はする[*1]。
- renIDecls

2015-05-15 (Fri)

[Bunny] Renamer まだつづき…

- renExp for LamExp and ParExp
- putStrLn を primNames に追加 （Assump はまだ。名前のみ。）
- A.VarExp は Con だけじゃなく Var にも

つづきは、typeSigDecl の処理から（他にもクラス宣言なども）

[Bunny] Haskell はおもしろい！ でも…

「以上！」とはならなかった。

単純に難しいから、というわけではなく。

[*1] ただし、クラス宣言を特別扱いする必要はあるかも。クラス宣言中のメソッドについては、型宣言のみで定義がなくても可、とか。

慣れればかけるようになりそうな気もしたけど、それだけでは、C や Assembly を理解しているようには、理解できるようになる気がしなくて。

コンパイラを書きたいと思った動機についてのメモ。

2015-05-14 (Thu)

[Bunny] Renamer ...

今日はいくつか進んだような。つぎは LamExp の rename から。

```
("Main.$",[PVar "Main.l3.f",PVar "Main.l3.x"])
("Main..",[PVar "Main.l4.f",PVar "Main.l4.g"])
LamExp [VarExp (Name {orig_name = "x", qual_name = "", name_pos = (37,12), isConName = False})] (Fun
drive_semant: Non-exhaustive patterns in renExp.
```

そろそろ pretty printer 欲しいよね。Show 有能でかなり助かってるんだけども。

[Bunny] tqd.hs を書いたのは…

あのとき[*2] は、まだ Tiger book も読んでなくて、GHC のコードとか見ながら理解できるかな〜というような無謀なことを考えていたのだった。

どの後なんとか Tiger book 読みおわってから再度自作 Haskell コンパイラに着手して、

ようやく Parse できるようになったのが 2015 年 2 月末だった[*3]みたい。

夏頃には「うごいた！」と言いたい。

2015-05-13 (Wed)

[Bunny] Renamer, renExp for CaseExp

```
$ sample/drive_semant < <(cat testcases/tqd.hs testcases/minlib.hs)
Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"),(":","Prim.:"),("Show","Prim.Show"),
("Main.qsort",[PCon ("Prim.[]" :>: Forall [Star] ([] :=> TAp (TCon (Tycon "[]" (Kfun Star Star))) (T
("Main.qsort",[PCon ("Prim.:" :>: Forall [Star] ([] :=> TAp (TAp (TCon (Tycon "(->)" (Kfun Star (Kfu
renDecls for LetExp
("Main.l1.l0.smaller",[])
renDecls for LetExp
("Main.l1.l0.l0.l0.OK",[PVar "Main.l1.l0.l0.l0.l0.a"])
renDecls for LetExp
drive_semant: qname not found: True
```

[*2] http://uhideyuki.sakura.ne.jp/uDiary/?date=20121003#p01

[*3] http://uhideyuki.sakura.ne.jp/uDiary/?date=20150224

- transHoge を renHoge にリネーム（transHoge は Core 言語への翻訳にとっておく）
- renExp for IfExp and CaseExp

True, False が未定義なので上のように引っかかった。
Semant.hs そのものも Renamer.hs とかに変えてもいいかもしれない。
つづく。

2015-05-12 (Tue)

[Bunny] Renamer つづき、List Comprehensions

tqd.hs をかけると、つぎのところでとまる：

```
("Main.l1.l0.smaller",[])
transDecls for LetExp
("Main.l1.l0.l0.l0.OK",[PVar "Main.l1.l0.l0.l0.l0.a"])
IfExp (InfixExp (VarExp (Name {orig_name = "a", qual_name = "", name_pos = (5,42), isConName = False
drive_semant: Non-exhaustive patterns in transExp.
```

これは、`smaller = [a \ a ¡- xs, a ¡= x]`— を desugar しているところで、

```
let ok a = [a | a <= x, True]
in concatMap ok xs
```

となったあとに、`[a \ a ¡= x, True]`— がさらに desugar されて `if a <= x then [a \ True] else []`— になって、そこで if 式の処理が未実装なので止まったというところ。
昨日、State モナドを Strict 版にしたので、どこで止まるのかがわかりやすくなった。
明日は続きで、if, case の処理を実装していく。

2015-05-10 (Sun)

[Bunny] Renamer つづき

transPat の実装を足していく。昨日は transPats の名にしていたが誤解だった。
`qsort (x:xs) = ..` の左辺が次のように変換された：

```
PCon ("Prim.:" :>: Forall [Star] ([] :=> TAp (TAp (TCon (Tycon "(->)" (Kfun Star (Kfun Star Star))))
```

- `infixr 5 :` がまだ。いまはデフォルトで infixl 9 と解釈されているはず
- transExp と transPat に同じような fixty resolution がある。DRY!
- なぜ "Main.l0.l0.x" のように 2 つネストされたレベルになってるの？要確認。

auto-save-buffers と変換候補の競合

Anthy と auto-save-buffers を両方使っているとき、変換候補が mini-buffer にあるのに auto-save-buffers が "Wrote hogehoge" っていうので、候補がかき消される問題。

むかーしにもこれに遭遇して対処した気がするんだが。

2015-05-09 (Sat)

[Bunny] Renamer つづき

昨日のつづき。test6.hs のつぎに tqd.hs をターゲットに。

Absyn の VarExp は、qcname を表現しているのだが…

```
aexp2
  : qcname                                  { VarExp $1 }
  | literal                                 { LitExp $1 }
  | '(' texp ')'                            { ParExp $2 }
  | '(' tup_exprs ')'                       { TupleExp $2 }
  | '[' list ']'                            { $2 }
  | '_'                                     { WildcardPat }
```

この qcname は conid だったり varid だったりする。

```
qcname: qvar                                { $1 }
      | qcon                                { $1 }
```

これ、fexp aexp の aexp のところに VarExp がくるんだけど、`f [] = ...` の `[]` と `f x = ...` の `x` がおなじ形でやってくるので、Semant にきたときに Data Constructor なのか Variable なのかわからない。

Semant で文字列を検査してどっちか判定するのは筋が悪すぎるので、Name の中に表示するのがいいのではないかと思った。幸い、Name の作成は必ず mkName でやっているので、mkName を改造すればいけるはず。

→ Name に isConName フィールドを追加、mkName は mkVName or mkCName に置き換え done.

つづきは、(x:xs) を transPats するところから。

2015-05-08 (Fri)

今日も少しでもコード書く。

[Bunny] Fixty Resolution

いちおう、test6.hs が [TempBind] に変換されるところまで。

```
[("Main.x",Nothing,[([],Ap (Ap (Var "Main.+") (Lit (LitInt 3))) (Ap (Ap (Var "Main.*") (Lit (LitInt
```

test6.hs を処理できるようにするため、Infix 宣言だけで辞書に登録してるけど、これは後に改めるかもしれない。後に改めないといけないのは、これだけじゃないのですが。

あとできれいにしたくなるとして、当面は、この調子で tqd.hs が扱えるようになるまで拡張していく。

[Bunny] 夏までに疎通（目標）

6 月初旬にはサンプルコードが扱える範囲の Renamer を書いて、型推論と Core への翻訳を 6 月中。疎通を 8 月までをとりあえずの目標にしておきたい。

2015-05-07 (Thu)

よく割れる腹筋の鍛え方[*4] 10 回× 3 セットくらいからはじめて徐々に増やしていけばどうかとのこと。

すがやみつるさんのこんにちは統計学[*5]

[Bunny] Fixty Resolution

ほんの少しだけど書いた。

```
$ sample/drive_semant < testcases/test6.hs
Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"),(":","Prim.:"),("Show","Prim.Show"),
Module {modid = Nothing, exports = Nothing, body = ([],[FixSigDecl Infixl 6 [Name {orig_name = "+",
VarExp (Name {orig_name = "x", qual_name = "", name_pos = (3,1)})
RnState {rn_modid = "Main", rn_lvs = [Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"
drive_semant: lookupInfixOp
```

lv_prefix と ifxenv から答えを返せるな。

[*4] https://twitter.com/nakashima723/status/590491376321564672

[*5] http://www.m-sugaya.jp/python/

2015-05-01 (Fri)

[Studs] 縦線

- 2015 05 — 04

↑のように書いたときの縦線の、LaTeX における扱いがおかしい。

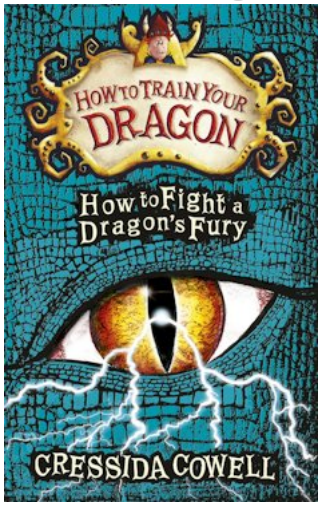
また、ふと見ると MathJax が動作していない状態になっていた。このサイトには https でアクセスするようにしてみたのだけど、https のサイトから MathJax には、そちらも https でアクセスするように指定しておかないと Javascript が信用されない（？）場合があるようだ。というわけで、ERB ファイルを書き換えて対処。

[Bunny] 結合性解決あたり

長いことコンパイラ作成のコーディングから離れてしまっていたので、思い出すところから。

```
$ sample/drive_semant < testcases/test6.hs
Level {lv_prefix = "Main", lv_dict = fromList [("()","Prim.()"),(":","Prim.:"),("Show","Prim.Show"),
VarExp (Name {orig_name = "x", qual_name = "", name_pos = (3,1)})
UnguardedRhs (InfixExp (InfixExp (LitExp (LitInteger 3 (3,5))) (Name {orig_name = "+", qual_name = "
drive_semant: Prelude.undefined
```

ヒックとドラゴン１２巻



原作者のページ[*6]によると、12 巻は今年の 9 月 8 日に出版されるらしい。邦訳が出るのはいつかな？

[*6] http://www.cressidacowell.co.uk/pages/blog_01/blog_item.asp?Blog_01ID=210