Компилятор GHC языка Haskell: теория языков программирования в работе

Лекция 2. Порядок компиляции одного модуля: от лексического анализа до кодогенерации

В. Н. Брагилевский

31 марта 2018 г.

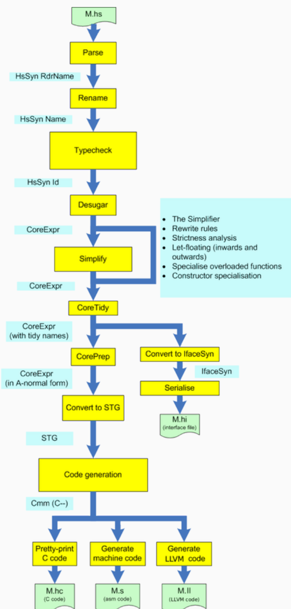
Computer Science клуб (Санкт-Петербург)

Институт математики, механики и компьютерных наук
имени И. И. Воровича, Южный федеральный университет (Ростов-на-Дону)

Компилятор GHC (`compiler`)

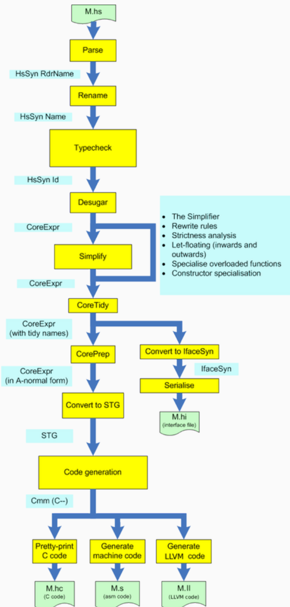
- Менеджер компиляции (что и в каком порядке компилировать)
- Компилятор одного модуля (The Haskell Compiler, `Hsc`)
- Драйвер (композиция `Hsc` с другими компонентами компилятора, такими как препроцессор C, ассемблер или линкер)

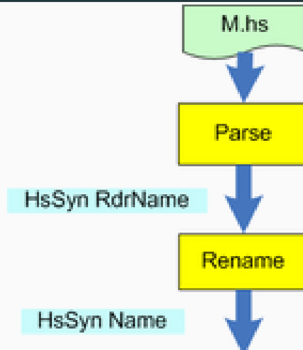
Компиляция одного модуля: общая схема



1. Разбор текста модуля (лексический и синтаксический анализ).
2. Разрешение имён.
3. Проверка типов.
4. Удаление синтаксического сахара и преобразование в Core.
5. Оптимизация.
6. Кодогенерация.

Компиляция модуля (1)





- HsSyn — дерево разбора (AST, Abstract Syntax Tree).
- RdrName, Name — имена с дополнительной информацией.

AST кода на Haskell (HsSyn) — модуль

```haskell
data HsModule name
  = HsModule {
    hsmodName :: Maybe
                 (Located ModuleName),
    hsmodExports :: Maybe
                 (Located [LIE name]),
    hsmodImports :: [LImportDecl name],
    hsmodDecls :: [LHsDecl name],
    -- ...
   }
```

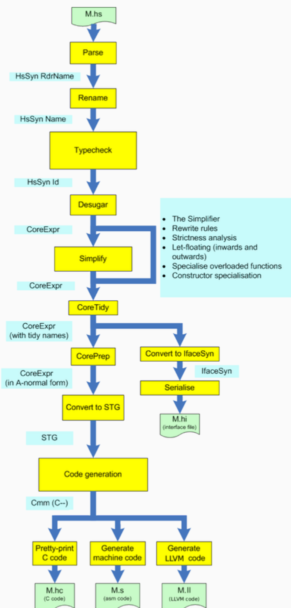
AST кода на Haskell (HsSyn) — объявления

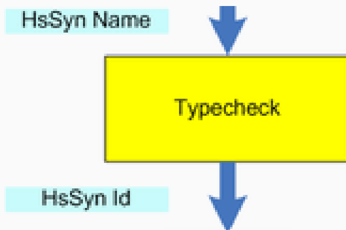
```haskell
type LHsDecl id = Located (HsDecl id)

data HsDecl id
  = TyClD         (TyClDecl id)
  | InstD         (InstDecl  id)
  | DerivD        (DerivDecl id)
  | ValD          (HsBind id)
  | SigD          (Sig id)
  | DefD          (DefaultDecl id)
  | ForD          (ForeignDecl id)
  ...
```

```
...
| WarningD     (WarnDecls id)
| AnnD         (AnnDecl id)
| RuleD        (RuleDecls id)
| VectD        (VectDecl id)
| SpliceD      (SpliceDecl id)
| DocD         (DocDecl)
| RoleAnnotD   (RoleAnnotDecl id)
```

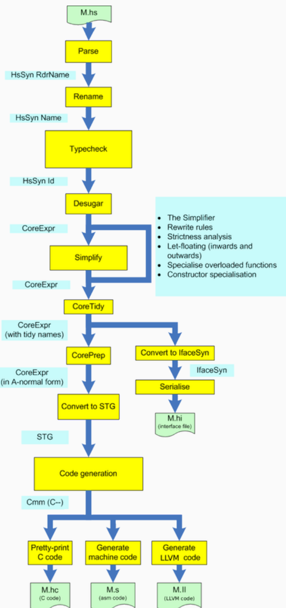
Компиляция модуля (2): проверка типов

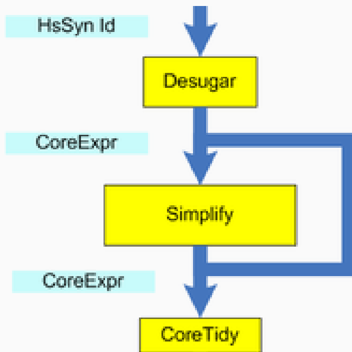




- Id — имя с информацией о типе (каждое выражение по результатам проверки типов имеет тип, указанный явно или выведенный).

Компиляция модуля (3): Core и его оптимизация





- Удаление синтаксического сахара — преобразование во внутренний язык (Core).
- Оптимизация.

Тип CoreExpr

```haskell
type CoreBndr = Var
type CoreExpr = Expr CoreBndr

data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion
```

```haskell
type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)
data AltCon = DataAlt DataCon
            | LitAlt  Literal
            | DEFAULT
data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]
```

Синтаксис Core: выражения и паттерны

t, e, u	$::=$	x	Переменные
	$\mid$	k	Литералы
	$\mid$	$\lambda x : \sigma . e \mid e\ u$	Абстракция и применение
	$\mid$	$\text{let } \overline{x : \tau = e} \text{ in } u$	Локальное связывание
	$\mid$	$\text{case } e \text{ of } \overline{p \to u}$	Выбор варианта
	$\mid$	$e \triangleright \gamma$	Преобразование типа значения (cast)
	$\mid$	τ	Тип
	$\mid$	$\lfloor \gamma \rfloor$	Приведение типа (coercion)
p	$::=$	$K\ \overline{c : \eta}\ \overline{x : \tau}$	Паттерны

- Надчёркивание — списки подтермов

Типы

$$
\begin{array}{lcl}
\tau,\ \kappa,\ \sigma,\ \phi & ::= & \\
& | & n \\
& | & \tau_1\ \tau_2 \\
& | & T\,\overline{\tau_i}^{\,i} \\
& | & \tau_1 \to \tau_2 \\
& | & \forall n.\tau \\
& | & \mathsf{lit} \\
& | & \tau \triangleright \gamma \\
& | & \gamma
\end{array}
$$

Оптимизация Core

- Упрощение (simplifier).
- Правила переписывания термов.
- Анализ строгости.
- Специализация перегруженных функций.
- Специализация конструкторов.
- …

Пример кода Core

```
f a b = a + b
main = print $ f 2 3
```

Пример кода Core

```haskell
f a b = a + b
main = print $ f 2 3
```

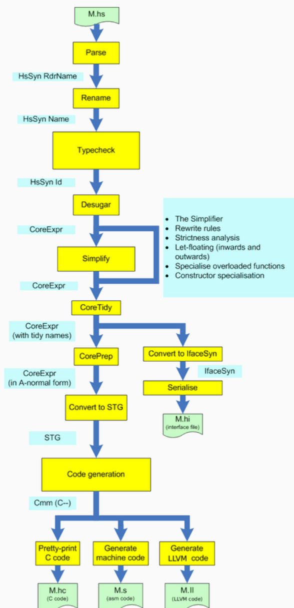
```
$ ghc -c ex.hs -ddump-simpl
```

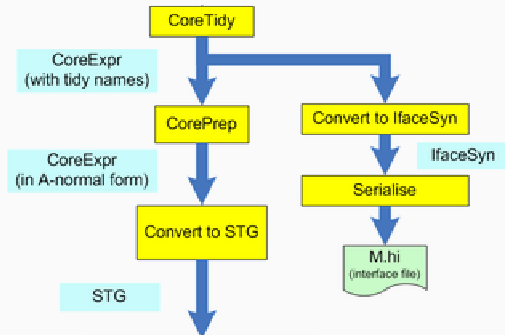
```
main :: IO ()
[GblId, Str=DmdType]
main =
 print
  @ Integer
  GHC.Show.$fShowInteger
  (+ @ Integer
       GHC.Num.$fNumInteger
       2 3)
```

Marlow и Peyton Jones, 2012:

> *In practice Core has been incredibly stable: over a 20-year time period we have added exactly one new major feature to Core (namely coercions and their associated casts). Over the same period, the source language has evolved enormously. We attribute this stability not to our own brilliance, but rather to the fact that Core is based directly on foundational mathematics:* ***bravo Girard!***

Компиляция модуля (4): подготовка к кодогенерации





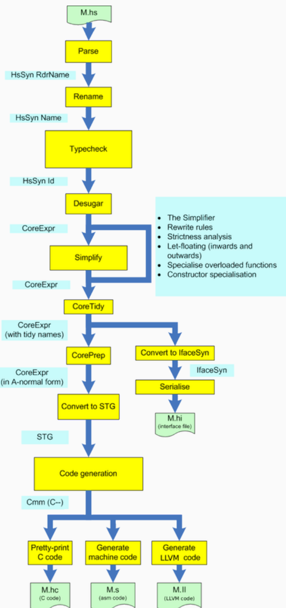
- Построение нормальной формы.
- Генерация интерфейсного файла (для межмодульной оптимизации).
- STG — Spineless Tagless G-machine

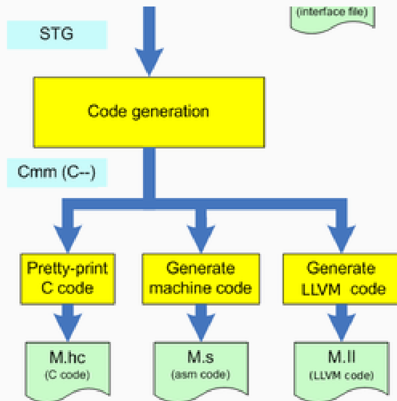
Фрагмент кода STG

```
sat_s10R :: GHC.Integer.Type.Integer
[LclId, Str=DmdType] =
    \u srt:SRT:[rp0 :-> GHC.Num.$fNumInteger] []
        let {
          sat_s10Q [Occ=Once] ::
                       GHC.Integer.Type.Integer
          [LclId, Str=DmdType] =
              NO_CCS GHC.Integer.Type.S#! [3#]; } in
        let {
          sat_s10P [Occ=Once] ::
                       GHC.Integer.Type.Integer
          [LclId, Str=DmdType] =
              NO_CCS GHC.Integer.Type.S#! [2#];
            } in  GHC.Num.+ GHC.Num.$fNumInteger
                            sat_s10P sat_s10Q;
```

```
Main.main :: GHC.Types.IO ()
[GblId, Str=DmdType] =
  \u srt:SRT:[0B :-> System.IO.print,
               rLs :-> GHC.Show.$fShowInteger,
               s10R :-> sat_s10R] []
  System.IO.print GHC.Show.$fShowInteger sat_s10R;
```

Компиляция модуля (5): кодогенерация





- Cmm — низкоуровневый императивный язык с явным стеком.

Фрагмент кода Cmm

```
I64[(old + 24)] = stg_bh_upd_frame_info;
I64[(old + 16)] = _c10Y::I64;
I64[Hp - 24] = GHC.Integer.Type.S#_con_info;
I64[Hp - 16] = 3;
_c111::P64 = Hp - 23;
I64[Hp - 8] = GHC.Integer.Type.S#_con_info;
I64[Hp] = 2;
_c112::P64 = Hp - 7;
R2 = GHC.Num.$fNumInteger_closure;
I64[(old + 48)] = stg_ap_pp_info;
P64[(old + 40)] = _c112::P64;
P64[(old + 32)] = _c111::P64;
call GHC.Num.+_info(R2) args: 48, res: 0, upd: 24;
```

Результаты кодогенерации

- Байт-код для интерпретации.
- Нативный код (на языке ассемблера).
- С-код.
- LLVM IR (промежуточный язык LLVM).

Способы расширения возможностей компилятора

- Определяемые пользователем правила переписывания термов.
- Плагины к компилятору.
- Компилятор как библиотека (GHC API).

Правила переписывания термов

```
{-# RULES "fold/build"
  forall k z (g::forall b.
              (a->b->b) -> b -> b) .
       foldr k z (build g) = g k z
 #-}
```

- Применение семантически корректных преобразований.

Плагины к компилятору

- Плагин — это один проход на этапе оптимизации — функция из Core в Core в отдельном файле.
- Подключение флагом компилятора или директивой в исходном коде.
- Компилятор динамически подключает преобразование и выполняет его.
- Аннотации плагинов — способ указания места применения преобразования.

Компилятор как библиотека (GHC API)

- Модульность позволяет подменять отдельные этапы компиляции.
- Каждый этап — функция, которую может вызвать пользователь в собственной программе.
- Компилятор целиком или частично может встраиваться в программу пользователя.

Важная книга

Simon L. Peyton Jones, The implementation of functional programming languages. 1987. 445 pp.

- Часть 1. Компиляция высокоуровневых функциональных языков.
- Часть 2. Редукция на графах.
- Часть 3. Усовершенствованная редукция на графах.

Список литературы

Marlow, Simon и Simon Peyton Jones (2012). “The Glasgow Haskell Compiler”. в: *The Architecture of Open Source Applications*. т. 2. гл. 5.

GHC Commentary: The Compiler. URL: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler`.