GHC source code diving

cyf, whz

June 1, 2017

1 Desugar

The Desugarer (compiler/deSugar/Desugar.hs) converts from the massive HsSyn type to GHC's intermediate language, CoreSyn. This Core-language data type is unusually tiny: just eight constructors. (-ddump-ds)

Generally speaking, the desugarer produces few user errors or warnings. But it does produce some. In particular, (a) pattern-match overlap warnings are produced here; and (b) when desugaring Template Haskell code quotations, the desugarer may find that THSyntax is not expressive enough. In that case, we must produce an error (compiler/deSugar/DsMeta.hs).

2 Core-to-Core transformations

2.1 The Core language

The Core language is GHC's central data types. Core is a very small, explicitly-typed, variant of System F. The exact variant is called System FC, which embodies equality constraints and coercions. The CoreSyn type, and the functions that operate over it, gets an entire directory compiler/coreSyn:

- compiler/coreSyn/CoreSyn.hs: the data type itself.
- compiler/coreSyn/PprCore.hs: pretty-printing.
- compiler/coreSyn/CoreFVs.hs: finding free variables.
- compiler/coreSyn/CoreSubst.hs: substitution.
- compiler/coreSyn/CoreUtils.hs: a variety of other useful functions over Core.
- compiler/coreSyn/CoreUnfold.hs: dealing with ”unfoldings”.
- compiler/coreSyn/CoreLint.hs: type-check the Core program. This is an incredibly-valuable consistency check, enabled by the flag -dcore-lint.

- compiler/coreSyn/CoreTidy.hs: part of the the CoreTidy pass (the rest is in compiler/main/TidyPgm.hs).
- compiler/coreSyn/CorePrep.hs: the CorePrep pass

Here is the entire Core type compiler/coreSyn/CoreSyn.hs:

```
type CoreExpr = Expr Var
data Expr b     -- ”b” for the type of binders,
  = 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

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))]
```

All of Haskell gets compiled through this tiny core.

Here are some notes about the individual constructors of Expr.

- Var represents variables. The Id it contains is essentially an OccName plus a Type; however, equality (==) on Ids is based only on their OccName's, so two Vars with different types may be (==)-equal.
- Lam is used for both term and type abstraction (small and big lambdas).
- Type appears only in type-argument positions (e.g. App (Var f) (Type ty)). To emphasise this, the type synonym Arg is used as documentation when we expect that a Type constructor may show up. Anything not called Arg should not use a Type constructor. Additional GHC Core uses so called type-lambdas, they are like lambdas, but instead of taking a real argument, they take a type instead. You should not confuse them with TypeFamilies, because type-lambdas are working on a value level, while type families are functions on the type level. The simplies example for a type-lambda usage is a polymorphic one: \ x → x. It will be represented in Core as A.id = \ (@ t_aeK) (x_aeG :: t_aeK) → x_aeG, where t_aeK is a *type argument*, so when specifying the argument of x_aeG we can refer to t_aeK. This is how polymorphism is represented in Core.

- Let handles both recursive and non-recursive let-bindings; see the the two constructors for Bind. The Let constructor contains both binders as well as the resulting expression. The resulting expression is the e in expression let x = r in e.

- Cast is used for an FC cast expression. Coercion is a synonym for Type.

- Tick is used to represent all the kinds of source annotation we support: profiling SCCs, HPC ticks, and GHCi breakpoints. Was named Note some time ago.

2.2 Core-to-Core optimization pipeline

After the source program has been typechecked it is desugared into GHC's intermediate language Core. The Core representation of a program is then optimized by a series of correctness preserving Core-to-Core passes. At the end of desugaring we run the simpleOptPgm function that performs some simple optimizations: eliminating dead bindings and inlining. The structure of the Core-to-Core pipeline is determined in the getCoreToDo function in the compiler/simplCore/SimplCore.hs module. Below is an ordered list of performed optimisations.

- **Static Argument Transformation**: tries to remove redundant arguments to recursive calls, turning them into free variables in those calls. Only enabled with -fstatic-argument-transformation. If run this pass is preceded with a "gentle" run of the simplifier.

- **Vectorisation**: run the Data Parallel Haskell vectoriser. Only enabled with -fvectorise.

- **Simplifier, gentle run**

- **Specialisation**: specialisation attempts to eliminate overloading.

- **Full laziness, 1st pass**: floats let-bindings outside of lambdas. This pass includes annotating bindings with level information and then running the float-out pass. In this first pass of the full laziness we don't float partial applications and bindings that contain free variables - this will be done by the second pass later in the pipeline.

- **Simplifier, main run**: run the main passes of the simplifier (phases 2, 1 and 0). Phase 0 is run with at least 3 iterations

- **Call arity**: attempts to eta-expand local functions based on how they are used. If run, this pass is followed by a 0 phase of the simplifier.

- **Demand analysis, 1st pass (a.k.a. strictness analysis)**: runs the demand analyser followed by worker-wrapper transformation and 0 phase of the simplifier. This pass tries to determine if some expressions are certain to be used and whether they will be used once or many times (cardinality analysis). We currently don't have means of saying that a binding is certain to be used many times. We can only determine that it is certain to be one-shot (ie. used only once) or probable to be one shot. Demand analysis pass only annotates Core with strictness information. This information is later used by worker/wrapper pass to perform transformations. CPR analysis is also done during demand analysis.

- **Full laziness, 2nd pass**: another full-laziness pass. This time partial applications and functions with free variables are floated out.
- **Common Sub-expression-elimination**: eliminates expressions that are identical.
- **Float in, 2nd pass**
- **Check rules, 1st pass**: this pass is not for optimisation but for troubleshooting the rules. It is only enabled with -frule-check flag that accepts a string pattern. This pass looks for rules beginning with that string pattern that could have fired but didn't and prints them to stdout.
- **Liberate case**: unrolls recursive functions once in their own RHS, to avoid repeated case analysis of free variables. It's a bit like the call-pattern specialisation but for free variables rather than arguments. Followed by a phase 0 simplifier run. Only enabled with -fliberate-case flag.
- **Call-pattern specialisation**: Only enabled with -fspec-constr flag.
- **Check rules, 2nd pass**
- **Simplifier, final**: final 0 phase of the simplifier.
- **Damand analysis, 2nd pass (a.k.a. late demand analysis)**: this pass consists of demand analysis followed by worker-wrapper transformation and phase 0 of the simplifier. The reason for this pass is that some opportunities for discovering strictness were not visible earlier; and optimisations like call-pattern specialisation can create functions with unused arguments which are eliminated by late demand analysis. Only run with -flate-dmd-anal.
- **Check rules, 3rd pass**

Simplifier is the workhorse of the Core-to-Core optimisation pipeline. It performs all the local transformations:

- constant folding
- applying the rewrite rules
- inlining

- case of case
- case of known constructor
- eta expansion and eta reduction
- combining adjacent casts
- pushing a cast out of the way of an application

Each run of the simplifier is assigned with a phase number: 2, 1 or 0. Phase numbers are used for control of interaction between the rules and INLINE/NOINLINE pragmas. There are many 0 phases because the simplifier is used to propagate the effects of other passes. There is also a special initial phase, which is used at the beginning of the pipeline by the ”gentle” simplifier runs.

Each single run of the simplifier consists of many iterations. Each iteration tries to apply all the optimisations. The simplifier run ends when a fixpoint is reached or when a predesignated number of iterations has been performed (the default is 4 and it can be controlled via the -fmax-simplifier-iterations flag).

The Core language is a functional language, and can be given the usual denotational semantics, but a direct operational interpretation also works for it, which facilitates the execution.

The operational model for Core requires a garbage-collected heap, which contains:

- *Data values*
- *Function values*
- *Thunks* (or suspensions), that represent suspended (yet unevaluated) values. Thunks are the implementation mechanism for Haskell's non-strict semantics.

The two most important operational intuitions about Core are:

- let bindings (and only let bindings) perform heap allocation.
- case expressions (and only case expressions) perform evaluation.

2.3 Polymorphism

As a strongly-typed language compiler, GHC infers the type of every expression and variable and takes advantage of the type information to generate better code. To implement the polymorphism mechanism, the program is decorated with type information and every program transformation must be sure to preserve it. For example, the composition function:

```
compose = /\a b c ->
        \f::(b->c) g::(a->b) x::a ->
        let y::b = g x in f y
```

The function takes three type parameters (a, b and c) and its value parameters f, g and x. A call of compose will be given three extra type arguments, which instantiate a, b and c just as the normal arguments instantiate f, g and x.

2.4 Inlining

Functional programs often consist of a myriad of small functions and functional programmers treat functions the way C programmers treat macros, so good inlining is crucial. Inlining removes some function-call overhead, and bring together code that was previously separated, which often exposes a cascade of new transformation opportunities, so in the simplifier inlining is implemented.

- **Inlining itself** replaces an occurrence of a let-bound variable by a copy of the right-hand side of its definition.
- **Dead code elimination** discards let bindings that are no longer used.
- **Beta reduction** replaces (\x→E)A by [x→A]E. And an analogous transformation deals with type applications.

There are two cases of inlining: **WHNFs** and **Non-WHNFs**. If a variable is bound to a *weak head normal form* (an atom, lambda abstraction or constructor application), then it can be inlined without risking the duplication of work and the trade-off is simply between code size and the benefit of inlining. Otherwise, inlining carries the risk of loss of sharing and hence the duplication of work (a transformation guaranteeing not to duplicate work is called *W-safe*).

In the case of WHNFs, atoms and constructor applications are always small enough to inline. (constructor applications must have atomic arguments.) Functions can be large so the size (in syntax nodes) of the body of the function is computed and accordingly make a heuristic method (which is a bit complicated).

In the case of Non-WHNFs, attention focuses on how the variable is used. If the variable occurs just once then presumably it is safe to inline, so a simple occurrence analysis that records for each variable how many places it is used is performed. Actually this approach turns to be complicated because the simplifier tries to perform as many transformations as possible during a single pass over the program, which may change the number of occurrences of a variable. Thus the current solution is to do a great deal of book-keeping to keep occurrence information up to date. Another problem is that, for example,

```
let x = f 100
      g = \y -> x
      in (g a) + (g b)
```

where if simply replace x by (f 100) then the call to f makes g called, rather than sharing it among all calls to g. Thus the current solution is also conservative: never inline inside a lambda abstraction, which practically and statically sufficiency.

2.5 Transforming conditionals

Most compilers have special rules to optimise conditionals. For example,

```
if (not x) then E1 else E2
```

No decent compiler would actually negate the value of x at runtime. Actually, after desugaring and inlining the definition of not, we get

```
case (case x of {True -> False; False -> True}) of
        True -> E1
        False -> E2
```

Here, the outer case scrutinises the value returned by the inner case, so the outer case can be moved inside the branches of the inner one:

```
case x of
        True -> case False of {True -> E1; False -> E2}
        False -> case True of {True -> E1; False -> E2}
```

and we obviously we can optimise this code into:

```
case x of
        True -> E2
        False -> E1
```

Conside more general cases:

```
case (case S of {True -> R1; False -> R2}) of
        True -> E1
        False -> E2
```

by let-bounding we get an opportunity of case-of-case transformation:

```
let e1 = E1; e2 = E2
in case S of
        True -> case R1 of {True -> e1; False -> e2}
        False -> R2 of {True -> e1; False ->e2}
```

we have given the example of not, which can be perfectly optimised, but in general cases we cannot guarantee that the newly-introduced bindings will be eliminated. It depends on R1 and R2. However, in many cases we CAN perform the elimination, and then inlining and simplification are usually performed.

2.6 Unboxed data types and strictness analysis

Because Core is non-strict, variables must be represented by a pointer to a possibly-unevaluated object. If evaluated, it will still therefore be represented by a pointer to a boxed value in the heap. In arithmetic operations boxing is possibly avoided because in many cases terms are evaluated immediately their components are evaluated and boxed, where a pair of boxing/unboxing are redundant. To implement boxing/unboxing and facilitate to expose boxing to transformation, instead of regarding the data types Int, Float and so on as primitive, the followings are defined using ordinary algebraic data declarations:

```
data Int = I# Int#
data Float = F# Float#
```

Here, Int# is the truly-primitive type of unboxed integers, and Float# is the type of unboxed floats, and previously-primitive + operation is expressed:

```
+ = \a b -> case a of
         I# a# -> case b of
                  I# b# -> case a# +# b# of
                           r# -> I# r#
```

where +# is the primitive addition operation on unboxed values.

GHC uses a rather simple strictness analyser, but it’s a bit difficult for us to dive into so we just neglect it...

2.7 Code motion

Three distinct kinds of let-floating transformations are useful to identify:

- *Floating inwards* moves bindings as far inwards as possible.
- *The full laziness transformation* floats selected bindings outside enclosing lambda abstractions.
- *Local transformations* ”fine-tune” the location of bindings.

Floating inwards is not difficult and obviously useful. Local transformations ”fine-tune” the placement of bindings, which consists of just three kinds:

- (let v=R in B) A ⇒ (let v=R in B A)
- case (let v=R in B) of {...} ⇒ let v=R in case B of {...}
- let x = let v=R1 in R2 in B ⇒ let v=R1 in let x=R2 in B

The first two transformations are always beneficial, because they do not change the number of allocations but do give other transformations more of a chance.

As for Full laziness, consider this example:

```
f = \xs -> letrec
          g = \y -> let n = length xs
                    in ... g ... n ...
          in ... g ...
```

Apparently ”let n = length xs” can be moved to the outer of the RHS of g. Such transformation is called full laziness.