Video: GHC Core language (14'04")

The `Core` type

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 | D

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b)
```

That's it. All of Haskell gets compiled through this tiny core.

`Expr` is parameterised over the type of its *binders*, `b`. This facility is used only rarely, and always temporarily; for example, the let-floater `SetLevels` pass attaches a binding level to every binder. By far the most important type is `CoreExpr`, which is `Expr` with `Var` binders. If you want to learn more about such AST-parametrization, I encourage you to read a blog post about it: http://blog.ezyang.com/2013/05/the-ast-typing-problem .

Binder is used (as the name suggest) to bind a variable to an expression. The `Expr` data type is parametrized by the binder type. The most common one is the `type CoreBndr = Var` where `Var` comes from compiler/basicTypes/Var.hs, which in fact is a `Name` with some extra informations attached (like types).

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 `Id`s is based only on their `OccName`'s, so *two* `Var`*s 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 specyfying 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`.

- `Case` expressions need more explanation.

- `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.

Case expressions

Case expressions are the most complicated bit of `Core`. In the term `Case scrut case_bndr res_ty alts`:

- `scrut` is the scrutinee
- `case_bndr` is the **case binder** (see notes below)
- `res_ty` is the type of the entire case expression (redundant once

FC is in HEAD -- was for GADTs)

- `alts` is a list of the case alternatives

A case expression can scrutinise

- **a data type** (the alternatives are `DataAlt`s), or
- **a primitive literal type** (the alternatives are `LitAlt`s), or
- **a value of any type at all** (if there is one `DEFAULT` alternative).

A case expression is **always strict**, even if there is only one alternative, and it is `DEFAULT`. (This differs from Haskell!) So

```
case error "urk" of { DEFAULT -> True }
```

will call `error`, rather then returning `True`.

The `case_bndr` field, called the **case binder**, is an unusual feature of GHC's case expressions. The idea is that *in any right-hand side, the case binder is bound to the value of the scrutinee*. If the scrutinee was always atomic nothing would be gained, but real expressiveness is added when the scrutinee is not atomic. Here is a slightly contrived example:

```
case (reverse xs) of y
  Nil        -> Nil
  Cons x xs -> append y y
```

(Here, "`y`" is the case binder; at least that is the syntax used by the Core pretty printer.) This expression evaluates `reverse xs`; if the result is `Nil`, it returns `Nil`, otherwise it returns the reversed list appended to itself. Since the returned value of `reverse xs` is present in the implementation, it makes sense to have a name for it!

The most common application is to model call-by-value, by using `case` instead of `let`. For example, here is how we might compile the call `f (reverse xs)` if we knew that `f` was strict:

```
case (reverse xs) of y { DEFAULT -> f y }
```

Case expressions have several invariants

- The `res_ty` type is the same as the type of any of the right-hand sides (up to refining unification -- coreRefineTys in compiler/types/Unify.hs -- in pre-FC).

- If there is a `DEFAULT` alternative, it must appear first. This makes finding a `DEFAULT` alternative easy, when it exists.

- The remaining non-DEFAULT alternatives must appear in order of
  - tag, for `DataAlt`s
  - lit, for `LitAlt`s

  This makes finding the relevant constructor easy, and makes comparison easier too.

- The list of alternatives is **always exhaustive**, meaning that it covers **all reachable cases**. Note, however, that an "exhausive" case does not necessarily mention all constructors:

```
data Foo = Red | Green | Blue

...case x of
        Red   -> True
        other -> f (case x of
                        Green -> ...
                        Blue  -> ... )
```

The inner case does not need a `Red` alternative, because x can't be `Red` at that program point. Furthermore, GADT type-refinement might mean that some alternatives are not reachable, and hence can be discarded.

Shadowing

One of the important things when working with Core is that variable shadowing is allowed. In other words, it is possible to come across a definition of a variable that has the same name (`realUnique`) as some other one that is already in scope. One of the possible ways to deal with that is to use `Subst` (substitution environment from compiler/coreSyn/CoreSubst.hs), which maintains the list of variables in scope and makes it possible to clone (i.e. rename) only the variables that actually capture names of some earlier ones. For some more explanations about this approach see Secrets of the Glasgow Haskell Compiler inliner (JFP'02) (section 4 on name capture).

Human readable Core generation

If you are interested in the way Core is translated into human readable form, you should check the sources for compiler/coreSyn/PprCore.hs. It is especially usefull if you want to see how the Core data types are being build, especially when there is no Show instance defined for them.

Last modified on Dec 23, 2014 9:22:44 PM