The LVM assembler library

Daan Leijen
daan@cs.uu.nl, http://www.cs.uu.nl/~daan

June 12, 2018

1 Library structure

The Core assembler library consists of:

- `common`. Common modules, for example for identifiers and binary files.
- `lvm`. Low level modules for handling LVM files and instructions.
- `asm`. Assembler language, a bit like STG language.
- `core`. Core language, enriched lambda calculus.
- `parsec`. Parser combinator library.

You can plug into the library at the Lvm, Asm or Core level. For most applications, the Core level is the most suitable abstraction.

- Lvm. The Lvm binary level. You need to generate an `Lvm.LvmModule`. This contains items like declared constructors and (function) values. `LvmWrite.lvmWriteFile` emits a binary Lvm file. The values contain an instruction stream (`Instr`). Fortunately, the libraries `InstrResolve` and `InstrRewrite` can resolve local variables and perform peephole optimization.
- Asm. The assembler level. An `Asm.AsmModule` contains Asm expressions, a restricted form of lambda calculus. `AsmToLvm.asmToLvm` generates Lvm instructions and converts to an `LvmModule`.
- Core. The Core level. A `Core.CoreModule` contains Core expressions, an enriched lambda calculus. `CoreToAsm.coreToAsm` rewrites this into an `Asm.AsmModule`.

The module `core/Main` implements a simple compiler from Core expressions to Lvm modules and illustrates how to call the different modules.

2 The Module library

Central to all these modules is the `lvm/Module.Module` structure. Here is the definition:

```
data Module v   = Module{ moduleName     :: !Id
                        , moduleMajorVer :: !Int
                        , moduleMinorVer :: !Int
                        , moduleDecls    :: [Decl v]
                        }
```

A module contains the *minimal* information necessary to generate a binary LVM file. Declarations are defined as:

```
data Decl v
  = DeclValue     { declName :: Id, declAccess :: !Access
                  , valueEnc :: Maybe Id, valueValue :: v
                  , declCustoms :: ![Custom] }
  | DeclAbstract  { declName :: Id, declAccess :: !Access
                  , declArity :: !Arity, declCustoms :: ![Custom] }
  | DeclCon       { declName :: Id, declAccess :: !Access
                  , declArity :: !Arity, conTag :: !Tag
                  , declCustoms :: [Custom] }
  | DeclExtern    { declName :: Id, declAccess :: !Access
                  , declArity :: !Arity
                  , externType :: !String, externLink :: !LinkConv
                  , externCall  :: !CallConv, externLib  :: !String
                  , externName :: !ExternName, declCustoms :: ![Custom] }
  | DeclCustom    { declName :: Id, declAccess :: !Access
                  , declKind :: !DeclKind, declCustoms :: ![Custom] }
  | DeclImport    { declName :: Id, declAccess :: !Access
                  , declCustoms :: ![Custom] }

data Access
  = Defined  { accessPublic :: !Bool }
  | Imported { accessPublic :: !Bool
             , importModule :: Id, importName :: Id
             , importKind :: !DeclKind
             , importMajorVer :: !Int, importMinorVer :: !Int }
```

Each declaration contains an `Access`. A declaration is either defined in this module but the access can also designate this declaration as *imported*. This is useful for constructors and externals – an implementation has all declarations available as if they are locally declared and doesn't need two kinds of declarations. Only when an lvm file is written, they are treated differently. For values the situation is handled differently since (non-inlined) imported values normally don't contain their definition. The `DeclAbstract` declaration is used for those.

A value declaration is parameterized by the actual definition value `v`. This means that each pass of the compiler can use the *same* module structure but each time with different definition values. Here are the type definitions for each major pass.

```
type LvmModule  = Module [Instr]  -- List of instructions
type AsmModule  = Module Top      -- Top  == top level Asm expressions
type CoreModule = Module Expr     -- Expr == Core expressions

coreToAsm   :: CoreModule -> AsmModule
asmToLvm    :: AsmModule -> LvmModule
lvmToBytes  :: LvmModule -> Bytes
```

3 Identifiers

The biggest obstacle to useing these libraries from another front-end compiler is the representation of identifiers. The library expects two interfaces `common/Id` and `common/IdMap` to be implemented. The default implementations work well but may not be suitable the front-end compiler. The compiler needs to provide a wrapper that implements both modules in terms of its own representation of identifiers or it needs to translate compiler identifiers into library identifiers when translating into Core or Asm modules. The last approach is used by the Helium compiler.

4 Imports

The `DeclImport` declarations are unresolved import declarations. The `DeclImport` declarations can be resolved by a call to `LvmResolve.lvmResolve`. Each import declaration is than replaced by a normal declaration with an `Imported` access, and imported value declarations are replaced by an abstract declaration.

If the `DeclImport` has an `importKind` of `DeclKindModule`, the `importName` is ignored and all the items exported by that module are imported.

5 Custom values

Custom values are defined as:

```
data Custom
  = CustomInt   !Int
  | CustomBytes !Bytes
  | CustomName  Id
  | CustomLink  Id !DeclKind
  | CustomDecl  !DeclKind ![Custom]
  | CustomNothing

data DeclKind
  = DeclKindName
  | DeclKindKind
  | DeclKindBytes
  | DeclKindCode
  | DeclKindValue
```

```
  | DeclKindCon
  | DeclKindImport
  | DeclKindModule
  | DeclKindExtern
  | DeclKindExternType
  | DeclKindCustom !Id
  deriving Eq
```

Basic custom values are a `CustomNothing`, `CustomInt`, `CustomBytes` (or strings) or a `CustomName` (for static link time identifiers). A `CustomLink` establishes a link to another declaration. A `CustomDecl` is a local anonymous declaration. For example, a type signature can be attached to a value declaration by adding the following custom value:

```
CustomDecl (DeclKindCustom (idFromString "type"))
           [CustomBytes (bytesFromString "...")]
```

6 Core assembler syntax

6.1 Notational conventions

These notational conventions are used for presenting syntax:

production	$\rightarrow$	$[\,p\,]$	optional
	$\mid$	$\{p\}^*$	zero or more repetitions
	$\mid$	$\{p\}^+$	one or more repetitions
	$\mid$	$(\!\mid p\ \ q \mid\!)^*$	zero or more p seperated by q
	$\mid$	$p \mid q$	choice
	$\mid$	$p_{\langle q \rangle}$	difference: p except those in q
	$\mid$	`terminal`	terminals are in typewriter font
	$\mid$	$_{\mathrm{x}}$0D	hexadecimal character code
$production_{[lex]}$	$\rightarrow$		lexemes are drawn recursively from *lex*

6.2 General products

The syntax for general products (or tuples) is:

product	$\rightarrow$	`(` `@` *tag* `,` *arity* `)`
arity	$\rightarrow$	*int*
tag	$\rightarrow$	*varid* $\mid$ *int*

Note that the *tag* should either be an *evaluated* variable or an integer. For example, a constructor with tag 0 and arity 2 can be build as:

```
tuple x y    = (@0,2) x y
```

There is special syntax for tuples that always have a zero tag, and the above example is equivalent to:

```
tuple x y    = (x,y)
```

The generated Core expressions for general products use the `ConTag` to describe the constructor. For example, the tuple `(x,y)` is translated into:

```
Ap (Ap (Con (ConTag (Lit (LitInt 0)) 2)) (Var x)) (Var y)
```

One can match on general products too but no variables are allowed for the tag yet.

patproduct	→	`(` `@` *pattag* `,` *arity* `)`
pattag	→	*int*

For example:

```
first x = case x of (@0,2) a b -> a
```

or equivalently:

```
first x = case x of (a,b) -> a
```

A noteworthy feature of the above function is that it will return the first field of *any* constructor with tag 0 and more than 1 field, allthough this behaviour might change with future versions of the LVM.

6.3 Custom values in core assembly

The syntax for custom values in core assembler is:

attributes	→	`:` [`private` | `public`] *customs*	
customs	→	`[` ⦇ *custom* `,` ⦈* `]`	
custom	→	*int*	custom int
	|	*string*	custom bytes
	|	`nothing`	custom nothing

	$\mid$	`custom` *declkind customid*	custom link
	$\mid$	`custom` *declkind customs*	anonymous declaration

customid	$\rightarrow$	*id* $\mid$ *string*	
declkind	$\rightarrow$	*id*	custom kind
	$\mid$	*string*	custom kind
	$\mid$	*int*	standard kind by number

6.3.1 Toplevel values

Custom values for toplevel values can be given right after the function arguments:

value	$\rightarrow$	*variable* $\{varid\}^*$ $[\,attributes\,]$ `=` *expr*

Here is an example where the `id` function is given a type.

```
module Id where
id x : public [custom type ["forall a. a -> a"]] = x
```

There is special syntax for type signatures and the above example is equivalent to:

```
module Id where
id :: a -> a
id x : public = x
```

Furthermore, values can be made public by specifying a Haskell style export list:

```
module Id( id ) where
id :: a -> a
id x = x
```

6.3.2 Custom declarations

A toplevel custom declaration starts with the `custom` keyword:

customdecl	$\rightarrow$	`custom` *declkind customid* $[\,attributes\,]$

For example, here is a kind declaration for a data type and an infix declaration:

```
custom "data" List : [custom kind ["*->*"]]
custom infix  "+"  : [left,5]
```

The assembler automatically adds kind and type signatures for data declarations.

```
data List a = Nil | Cons a (List a)
```

The above example is equivalent with[1]:

```
custom "data" List : [custom kind ["*->*]]
con Nil  : [custom type ["forall a. List a"]
           ,custom data List] = (@0,0)
con Cons : [custom type ["forall a. a -> List a -> List a"]
           ,custom data List] = (@1,2)
```

The visibility of a data type can be specified with a Haskell style export list. For example:

```
module Data( List(Cons,Nil) ) where ...
```

or even:

```
module Data( List(..) ) where ...
```

[1]except that `con` declarations are not supported (yet).