Module Feldspar

This module is used as a front-end to the Feldspar language. It re-exports from the internal modules.

```haskell
{-# LANGUAGE DataKinds #-}
module Feldspar (module Feldspar.FrontEnd.Interface
  ,Data,Int32,Num (..),String) where
import qualified Prelude
import Prelude (String,Num (..))


import Feldspar.FrontEnd.Interface
import qualified Feldspar.FrontEnd.AST as AST
import qualified Feldspar.Types as Types


import Feldspar.Annotations ()


type Data a = AST.Data a ( String )
type Int32 = Types.Int32
```

Module Annotations

The module containing the type classes and the functions to facilitate injecting, projecting and preserving the annotations.

```haskell
{-# LANGUAGE TypeFamilies #-}
module Annotations where
import qualified Prelude
import Prelude (Maybe (..))

  -- injecting annotations into data
class Inj t where
  type Ann t
  inj :: Ann t → t → t

  -- projecting the stored annotations
class Inj t ⇒ Annotatable t where
  prj :: t → Maybe ((Ann t,t))
```

```
-- preserving the annotations
preserve :: ∀ t_Hi t_Lo.
  (Annotatable t_Hi,Inj t_Lo
  ,Ann t_Hi~Ann t_Lo) ⇒
  t_Hi → (t_Hi → t_Lo) → t_Lo
preserve e_Hi f = case prj e_Hi of
  Just (ann,e'_Hi) → inj ann (f e'_Hi)
  Nothing → f e_Hi
```

Module BX

This module contains the code for our semantic bidirectionalization algorithm, described in the thesis.

```
-- the code is omitted
```

Module Feldspar.Compiler

This module is used as a front-end to the Feldspar compiler. It re-exports from the internal modules.

```
module Feldspar.Compiler (icompile,scompile,IO) where
import Feldspar.Compiler.Compiler
```

Module Feldspar.Types

This module contains the declaration of the built-in types in Pico-Feldspar. It also includes the code defining singlton types and the utility functions for promotion and demotion of the built-in types.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ScopedTypeVariables #-}
module Feldspar.Types where
import qualified Prelude
import Prelude (Eq (..))
```

```
  -- the built-in types
  -- it is usually used in the promoted form
data Types = Int32 | Bool
  deriving Eq

  -- a GADT representation of a singleton type for
  -- the built-in types
data SingTypes :: Types → *where
  SInt32 :: SingTypes Int32
  SBool :: SingTypes Bool

  -- overloaded function to demote singletons
class SingT (n :: Types) where
  sing :: SingTypes n
instance SingT Int32 where
  sing = SInt32
instance SingT Bool where
  sing = SBool

  -- coversion from singleton types to the original
toTypes :: SingTypes n → Types
toTypes SInt32 = Int32
toTypes SBool = Bool

  -- overloaded function to demote singletons
  -- to the original
getType :: ∀ k n a.SingT n ⇒ k n a → Types
getType _ = toTypes (sing :: SingTypes n)

  -- an overloaded function to facilitate demotion
  -- using the type of the argument of a function
getTypeF :: ∀ k n a r.SingT n ⇒
  (k n a → r) → SingTypes n
getTypeF _ = sing :: SingTypes n
```

Module Feldspar.Annotations

In this module, the type classes defined in the module *Annotations* are derived for the main data types in Pico-Feldspar.

```
-- the code is omitted
```

Module Feldspar.AnnotationUtils

This module provides a set of utilities to work with annotations, e.g., removing all the annotations from an AST *stripAnn* or annotating every single node in an AST with the value *False*.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleInstances #-}
module Feldspar.AnnotationUtils where
import qualified Prelude as P
import Prelude (Maybe (..),map,($),(.))
```

```
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
```

```
import Annotations (Inj (..))
import Feldspar.Annotations ()
```

-- removing all the annotations
$stripAnn :: Data\ a\ ann \rightarrow Data\ a\ ann'$
$stripAnn\ (Var\ \ \ \ \ x) = Var\ \ \ x$
$stripAnn\ (Lit\_Int\ \ i) = Lit\_Int\ i$
$stripAnn\ (Lit\_Bool\ b) = Lit\_Bool\ b$
$stripAnn\ (Not\ \ \ \ \ e) = Not\ (stripAnn\ e)$
$stripAnn\ (Add\ \ \ \ e_1\ e_2) =$
$\quad Add\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (Sub\ \ \ \ e_1\ e_2) =$
$\quad Sub\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (Mul\ \ \ \ e_1\ e_2) =$
$\quad Mul\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (Eq\_Int\ e_1\ e_2) =$
$\quad Eq\_Int\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (LT\_Int\ e_1\ e_2) =$
$\quad LT\_Int\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (And\ \ \ \ e_1\ e_2) =$
$\quad And\ (stripAnn\ e_1)\ (stripAnn\ e_2)$
$stripAnn\ (If\ e_1\ e_2\ e_3) =$
$\quad If\ (stripAnn\ e_1)\ (stripAnn\ e_2)\ (stripAnn\ e_3)$
$stripAnn\ (Ann\ \_\ e) = stripAnn\ e$

-- annotating each node in the output with `False`
$markAllF :: \forall\ a\ ann\ ann'\ r.$
$\quad (r\ ann' \rightarrow r\ P.Bool) \rightarrow$
$\quad (Data\ a\ ann \rightarrow r\ ann') \rightarrow$
$\quad (Data\ a\ P.Bool \rightarrow r\ P.Bool)$
$markAllF\ markAllr\ f = markAllr.f.stripAnn$

```
-- annotating each node with False
markAll :: ∀ a ann.Data a ann
           → Data a P.Bool
markAll (Var      x) = Ann P.False $
  Var     x
markAll (Lit_Int  i) = Ann P.False $
  Lit_Int i
markAll (Lit_Bool b) = Ann P.False $
  Lit_Bool b
markAll (Not      e) = Ann P.False $
  Not (markAll e)
markAll (Add     e1 e2) = Ann P.False $
  Add (markAll e1) (markAll e2)
markAll (Sub     e1 e2) = Ann P.False $
  Sub (markAll e1) (markAll e2)
markAll (Mul     e1 e2) = Ann P.False $
  Mul (markAll e1) (markAll e2)
markAll (Eq_Int  e1 e2) = Ann P.False $
  Eq_Int (markAll e1) (markAll e2)
markAll (LT_Int e1 e2) = Ann P.False $
  LT_Int (markAll e1) (markAll e2)
markAll (And     e1 e2) = Ann P.False $
  And (markAll e1) (markAll e2)
markAll (If e1 e2 e3) = Ann P.False $
  If (markAll e1) (markAll e2) (markAll e3)
markAll (Ann _ e) =
  markAll e
```

```
-- helper function
annCond :: ∀ k.Inj k ⇒
  Maybe (Ann k) → k → k
annCond (Just ann) e = inj ann e
annCond Nothing e = e
```

```
-- pushing down the annotation, so the unannotated
-- nodes inherit the parent's annotation
class PushDown t where
  pushDown :: (Maybe (Ann t)) →
    t → t
```

```haskell
  -- pushing down the annotations for functions
instance PushDown r ⇒
  PushDown (Data a ann → r) where
  pushDown ann f = pushDown ann.f
```

```haskell
  -- pushing down the annotation for terms of
  -- type Data a ann
instance PushDown (Data a ann) where
  pushDown ann (Var      x) = annCond ann $
    Var     x
  pushDown ann (Lit_Int  i) = annCond ann $
    Lit_Int i
  pushDown ann (Lit_Bool b) = annCond ann $
    Lit_Bool b
  pushDown ann (Not      e) = annCond ann $
    Not (pushDown ann e)
  pushDown ann (Add    e1 e2) = annCond ann $
    Add (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Sub    e1 e2) = annCond ann $
    Sub (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Mul    e1 e2) = annCond ann $
    Mul (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Eq_Int e1 e2) = annCond ann $
    Eq_Int (pushDown ann e1) (pushDown ann e2)
  pushDown ann (LT_Int e1 e2) = annCond ann $
    LT_Int (pushDown ann e1) (pushDown ann e2)
  pushDown ann (And    e1 e2) = annCond ann $
    And (pushDown ann e1) (pushDown ann e2)
  pushDown ann (If e1 e2 e3) = annCond ann $
    If (pushDown ann e1) (pushDown ann e2)
      (pushDown ann e3)
  pushDown _ (Ann ann e) =
    pushDown (Just ann) e
```

```
  -- pushing down the annotation for terms of
  -- type Exp_C ann
instance PushDown (Exp_C ann) where
  pushDown ann (Var_C x)     = annCond ann $
    Var_C x
  pushDown ann (Num i)       = annCond ann $
    Num i
  pushDown ann (Infix e1 x e2) = annCond ann $
    Infix (pushDown ann e1) x (pushDown ann e2)
  pushDown ann (Unary x e) = annCond ann $
    Unary x (pushDown ann e)
  pushDown _ (Ann_ExpC ann e) =
    pushDown (Just ann) e
```

```
  -- pushing down the annotation for terms of
  -- type Stmt ann
instance PushDown (Stmt ann) where
  pushDown ann (If_C e stmts1 stmts2) = annCond ann $
    If_C (pushDown ann e) (pushDown ann `map` stmts1)
      (pushDown ann `map` stmts2)
  pushDown ann (Assign x e)       = annCond ann $
    Assign x (pushDown ann e)
  pushDown ann (Declare t x)      = annCond ann $
    Declare t x
  pushDown _ (Ann_Stmt ann stmt) =
    pushDown (Just ann) stmt
```

```
  -- pushing down the annotation for terms of
  -- type Func ann
instance PushDown (Func ann) where
  pushDown ann (Func x vs stmts) = Func x vs $
    pushDown ann `map` stmts
```

Module Feldspar.BX

This module provides the necessary functions to bidirectionalize the transformation from EDSL to C code by composing the bidirectionalization of each smaller transformations in between.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
module Feldspar.BX where
import qualified Prelude as P
import Prelude (String,Either (..),Maybe (..),Eq (..)
  ,Read (..),Monad (..),map,zip,(.),($))
import Data.Foldable (toList)
```

```
import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.Compiler.BXCompiler (BXable (..))
import Feldspar.BackEnd.BXPretty (putPretty)
import Feldspar.Compiler.Compiler (toFunc,compile)
import Feldspar.BackEnd.Pretty (Pretty (..))
import Feldspar.AnnotationUtils (PushDown (..))
import Annotations (Inj (..))
```

```
  -- zipping similiar AST with different Annotations
class ZipData t t' where
  zipData :: t -> t' -> [(Ann t,Ann t')]
instance (SingT a,ZipData r r'
  ,Ann r~ann,Ann r'~ann') =>
  ZipData (Data a ann -> r)
                  (Data a ann' -> r') where
  zipData f g = zipData
    (f $ Var $ VarT "_x" sing)
    (g $ Var $ VarT "_x" sing)
instance ZipData (Data a ann) (Data a ann') where
  zipData d d' = zip (toList d) (toList d')
```

```
  -- putting back changes up to the src-loc
putAnn :: ∀ t t'.
  (PushDown t',BXable t,ZipData t t'
  ,Ann t~P.Bool) ⇒
  P.Bool → (t' → t) → t' → String →
  Either String [Ann t']
putAnn cn markA d src = do
  let dS = pushDown Nothing d
  let dM = markA d
  dU ← put cn dM src
  return [s | (b,s) ← zipData dU dS,b]
```

```
  -- putting back changes up to the high-level AST
put :: ∀ b.
    (Eq (Ann b),Read (Ann b),
      Pretty (Ann b),BXable b) ⇒
       P.Bool → b → String →
       Either String b
put b s v' = do
  let s' = (toFunc.compile 0) s
  let ps' = if b
    then pushDown Nothing s'
    else s'
  v ← putPretty ps' v'
  putCompile 0 s v
```

Module Feldspar.FrontEnd.AST

This module, provides the type-safe representation (via GADTs) of the high-level language.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
module Feldspar.FrontEnd.AST where
import qualified Prelude as P
import Feldspar.Types
```

```
-- AST of the EDSL (high-level)
data Data (a :: Types) ann where
  Var :: VarT a → Data a ann
  Lit_Int :: P.Int → Data Int32 ann
  Add :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Sub :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Mul :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Eq_Int :: Data Int32 ann → Data Int32 ann → Data Bool ann
  LT_Int :: Data Int32 ann → Data Int32 ann → Data Bool ann
  Lit_Bool :: P.Bool → Data Bool ann
  Not :: Data Bool ann → Data Bool ann
  And :: Data Bool ann → Data Bool ann → Data Bool ann
  If :: Data Bool ann → Data a ann → Data a ann → Data a ann
```

```
Ann :: ann → Data a ann → Data a ann
```

```
-- Variables
data VarT t = VarT P.String (SingTypes t)
```

Module Feldspar.FrontEnd.Interface

This module, provides some utility functions to program in the high-level language.

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE DataKinds #-}
module Feldspar.FrontEnd.Interface where
import qualified Prelude
import Prelude (Num (..),Int,($),Show,String)
import Feldspar.FrontEnd.AST
import Feldspar.Types
```

```
instance Num (Data Int32 ann) where
  fromInteger i = Lit_Int $ fromInteger i
  (+) = Add
  (−) = Sub
  (∗) = Mul
  signum x = condition (x < 0)
    (−1)
    (condition (x == 0) 0 1)
  abs x = (signum x) ∗ x
```

$$(==) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$(==) = \mathit{Eq}_{\mathit{Int}}$$

$$(<) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to$$
$$\qquad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$(<) = \mathit{LT}_{\mathit{Int}}$$

$$(>) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to$$
$$\qquad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$e_1 > e_2 = \neg\ \$\ e_1 < e_2$$

$$(\leqslant) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$e_1 \leqslant e_2 = (e_1 < e_2) \wedge (e_1 == e_2)$$

$$(\geqslant) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Int32}\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$e_1 \geqslant e_2 = (e_1 > e_2) \wedge (e_1 == e_2)$$

$$\mathit{true} :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$\mathit{true} = \mathit{Lit}_{\mathit{Bool}}\ \mathit{Prelude.True}$$

$$\mathit{false} :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$\mathit{false} = \mathit{Lit}_{\mathit{Bool}}\ \mathit{Prelude.False}$$

$$\neg :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$\neg = \mathit{Not}$$

$$(\wedge) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$(\wedge) = \mathit{And}$$

$$(\vee) :: \forall\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to \mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ \mathit{Bool}\ \mathrm{ann}$$
$$x \vee y = \neg\ ((\neg\ x) \wedge (\neg\ y))$$

$$\mathit{condition} :: \forall\ a\ \mathrm{ann}.\mathit{Data}\ \mathit{Bool}\ \mathrm{ann} \to \mathit{Data}\ a\ \mathrm{ann} \to$$
$$\quad \mathit{Data}\ a\ \mathrm{ann} \to \mathit{Data}\ a\ \mathrm{ann}$$
$$\mathit{condition} = \mathit{If}$$

Module Feldspar.FrontEnd.Derivings

In this module, the type classes *Functor*, *Foldable* and *Traversable* are derived for the high-level AST.

```
-- the code is omitted
```

Module Feldspar.Compiler.Compiler

This module, contains the main code for compiling the high-level AST to C code.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
module Feldspar.Compiler.Compiler where
import qualified Prelude as P
import Prelude ((.),Show (..),putStrLn,IO
  ,Int,String,(++),(+),Monad (..))
import Control.Monad.State (State,put,get
  ,evalState)


import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
import Feldspar.BackEnd.Pretty


import Feldspar.Annotations


  -- the monadic function to compile the
  -- the high-level AST to a pair containing
  -- an expression containing the returned
  -- value and a list of statements; the
  -- state contains a counter to generate
  -- fresh variables
compileM :: SingT a => Data a ann ->
  State Int (Exp_C ann,[Stmt ann])
```

```
compileM (Var (VarT v _)) =
  return (Var_C v,[])
compileM (Lit_Int x) =
  return (Num x,[])
compileM (Lit_Bool P.True) =
  return (Var_C "true",[])
compileM (Lit_Bool P.False) =
  return (Var_C "false",[])
```

```
compileM (Add e_1 e_2) = do
  (e_C1,st_1) ← compileM e_1
  (e_C2,st_2) ← compileM e_2
  return (Infix e_C1 "+" e_C2
    ,st_1 ++ st_2)
```

```
compileM (Sub e_1 e_2) = do
  (e_C1,st_1) ← compileM e_1
  (e_C2,st_2) ← compileM e_2
  return (Infix e_C1 "-" e_C2
    ,st_1 ++ st_2)
```

```
compileM (Mul e_1 e_2) = do
  (e_C1,st_1) ← compileM e_1
  (e_C2,st_2) ← compileM e_2
  return (Infix e_C1 "*" e_C2
    ,st_1 ++ st_2)
```

```
compileM (Eq_Int e_1 e_2) = do
  (e_C1,st_1) ← compileM e_1
  (e_C2,st_2) ← compileM e_2
  return (Infix e_C1 "==" e_C2
    ,st_1 ++ st_2)
```

```
compileM (LT_Int e_1 e_2) = do
  (e_C1,st_1) ← compileM e_1
  (e_C2,st_2) ← compileM e_2
  return (Infix e_C1 "<" e_C2
    ,st_1 ++ st_2)
```

```
compileM (And e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "&&" eC2
    ,st1 ++ st2)


compileM (Not e1) = do
  (eC1,st1) ← compileM e1
  return (Unary "!" eC1
    ,st1)


compileM e@(If e1 e2 e3) = do
  i ← get
  put (i + 1)
  let v = "v" ++ (show i)
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  (eC3,st3) ← compileM e3
  return
    (VarC v
    ,st1 ++
      [Declare (getType e) v
      ,IfC eC1
        (st2 ++ [Assign v eC2])
        (st3 ++ [Assign v eC3])])


compileM e = preserve e compileM


  -- overloaded function to compile
  -- regardless of AST being parametric
class Inj t ⇒
  Compilable t where
  compileF :: ([Var],t) →
    State Int
    ([Var],Types
    ,ExpC (Ann t)
    ,[Stmt (Ann t)])
```

```haskell
  -- a parametric AST is first applied to
  -- a fresh variable with the right type
  -- and then it is compiled
instance (SingT a, Compilable r) =>
      Compilable (Data a ann' -> r) where
  compileF (ps, f) = do
    i <- get
    put (i + 1)
    let v = "v" ++ (show i)
      a = Var (VarT v (getTypeF f))
      r = f a
    compileF ((ps ++ [(v, getType a)]), r)


  -- a non-parametric AST is compiled in
  -- the normal way defined in compileM
instance SingT a =>
  Compilable (Data a ann) where
  compileF (ps, d) = do
    (e, sts) <- compileM d
    return (ps, getType d, e, sts)


  -- coversion to Func
toFunc :: ([Var], Types, Exp_C ann, [Stmt ann]) ->
  Func ann
toFunc (ps, ty, exp_C, stmts) =
  Func "test" (ps ++ [("*out", ty)])
  (stmts ++ [Assign "*out" exp_C])


  -- running the state monad with a seed
compile :: Compilable a => Int ->
  a -> ([Var], Types, Exp_C (Ann a)
    , [Stmt (Ann a)])
compile seed d = evalState (compileF ([], d)) seed


  -- an interface to the compiler
scompile :: (Compilable a, Pretty (Ann a)) =>
  a -> String
scompile = show.pretty.toFunc.(compile 0)
```

```
  -- an interface to the compiler
icompile :: (Compilable a,Pretty ( Ann a )) ⇒
  a → IO ()
icompile = putStrLn.scompile
```

Module Feldspar.Compiler.Compiler

This module, contains the main code for compiling the high-level AST to C code.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE TypeSynonymInstances #-}
 {-# LANGUAGE FlexibleInstances #-}
 {-# LANGUAGE FlexibleContexts #-}
module Feldspar.Compiler.Compiler where
import qualified Prelude as P
import Prelude ((.),Show (..),putStrLn,IO
  ,Int,String,(++),(+),Monad (..))
import Control.Monad.State (State,put,get
  ,evalState)


import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
import Feldspar.BackEnd.Pretty


import Feldspar.Annotations


  -- the monadic function to compile the
  -- the high-level AST to a pair containing
  -- an expression containing the returned
  -- value and a list of statements; the
  -- state contains a counter to generate
  -- fresh variables
compileM :: SingT a ⇒ Data a ann →
  State Int (Exp_C ann ,[Stmt ann ])
```

```
compileM (Var (VarT v _)) =
  return (Var_C v,[])
compileM (Lit_Int x) =
  return (Num x,[])
compileM (Lit_Bool P.True) =
  return (Var_C "true",[])
compileM (Lit_Bool P.False) =
  return (Var_C "false",[])

compileM (Add e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "+" eC2
    ,st1 ++ st2)

compileM (Sub e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "-" eC2
    ,st1 ++ st2)

compileM (Mul e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "*" eC2
    ,st1 ++ st2)

compileM (Eq_Int e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "==" eC2
    ,st1 ++ st2)

compileM (LT_Int e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "<" eC2
    ,st1 ++ st2)
```

```
compileM (And e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "&&" eC2
    ,st1 ++ st2)

compileM (Not e1) = do
  (eC1,st1) ← compileM e1
  return (Unary "!" eC1
    ,st1)

compileM e@(If e1 e2 e3) = do
  i ← get
  put (i + 1)
  let v = "v" ++ (show i)
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  (eC3,st3) ← compileM e3
  return
    (VarC v
    ,st1 ++
      [Declare (getType e) v
      ,IfC eC1
        (st2 ++ [Assign v eC2])
        (st3 ++ [Assign v eC3])])

compileM e = preserve e compileM

  -- overloaded function to compile
  -- regardless of AST being parametric
class Inj t ⇒
  Compilable t where
  compileF :: ([Var],t) →
    State Int
    ([Var],Types
    ,ExpC (Ann t)
    ,[Stmt (Ann t)])
```

```haskell
  -- a parametric AST is first applied to
  -- a fresh variable with the right type
  -- and then it is compiled
instance (SingT a,Compilable r) ⇒
      Compilable (Data a ann' → r) where
  compileF (ps,f) = do
    i ← get
    put (i + 1)
    let v = "v" ++ (show i)
      a = Var (VarT v (getTypeF f))
      r = f a
    compileF ((ps ++ [(v,getType a)]),r)


  -- a non-parametric AST is compiled in
  -- the normal way defined in compileM
instance SingT a ⇒
  Compilable (Data a ann) where
  compileF (ps,d) = do
    (e,sts) ← compileM d
    return (ps,getType d,e,sts)


  -- coversion to Func
toFunc :: ([Var],Types,Exp_C ann,[Stmt ann]) →
  Func ann
toFunc (ps,ty,exp_C,stmts) =
  Func "test" (ps ++ [("*out",ty)])
  (stmts ++ [Assign "*out" exp_C])


  -- running the state monad with a seed
compile :: Compilable a ⇒ Int →
  a → ([Var],Types,Exp_C (Ann a)
    ,[Stmt (Ann a)])
compile seed d = evalState (compileF ([],d)) seed


  -- an interface to the compiler
scompile :: (Compilable a,Pretty (Ann a)) ⇒
  a → String
scompile = show.pretty.toFunc.(compile 0)
```

```
  -- an interface to the compiler
icompile :: (Compilable a,Pretty ( Ann a )) ⇒
  a → IO ()
icompile = putStrLn.scompile
```

Module Feldspar.Compiler.BXCompiler

This module contains the code to bidirectionalize the compile functions.

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
module Feldspar.Compiler.BXCompiler where
import qualified Prelude as P
import Prelude (String,Eq (..),Either (..),Int,const
  ,Monad (..),(.),tail,Show (..),(+)
  ,(++),($))
```

```
import BX
import Annotations
import Feldspar.Types
import Feldspar.BackEnd.AST
import Feldspar.FrontEnd.AST
import Feldspar.Compiler.Compiler
import Feldspar.FrontEnd.Derivings ()
import Feldspar.BackEnd.Derivings ()
```

```
  -- overloaded function to bidirectionalize
  -- instances of Compilable
class Compilable t ⇒ BXable t where
  putCompile :: Eq (Ann t) ⇒ Int →
    t → Func (Ann t) →
    Either String t
```

```
  -- Bidirectionalization done by bff_GUS_G_Gen
instance SingT a ⇒ BXable (Data a ann ) where
  putCompile i = bff_GUS_G_Gen
    (const (toFunc.compile i))
    (const () :: ∀ ann .Tuple_1 ann → ())
```

```
  -- Bidirectionalization done manually
instance (SingT a,BXable r
  ,Ann r~ ann ,Abstract r) ⇒
  BXable (Data a ann → r) where
  putCompile i f (Func x ps stmts) = do
    let n = "v" ++ (show i)
      vt = VarT n (getTypeF f)
      v = (Var vt)
      r = f v
    r′ ← putCompile (i + 1) r (Func x (tail ps) stmts)
    return $ λvv → abstract vt vv r′
```

```
  -- overloaded function to abstract over
  -- a variable and generate the parametric AST
class Abstract t where
  abstract :: ∀ a. VarT a →
    Data a (Ann t) → t → t
```

```
instance Abstract r ⇒
  Abstract (Data a ann → r) where
  abstract vt d f = abstract vt d.f
```

```
instance Abstract (Data a ann) where
  abstract (VarT v SBool) d
    e@(Var (VarT x SBool))
     | v == x = d
     | P.True = e
  abstract (VarT v SInt32) d
    e@(Var (VarT x SInt32))
     | v == x = d
     | P.True = e
  abstract _ _ e@(Var _) = e
```

```
abstract _ _ (Lit_Int i) =
  Lit_Int i
```

```
abstract _ _ (Lit_Bool b) =
  Lit_Bool b
```

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Not}\ e) = \\
&\quad \mathit{Not}\ (\mathit{abstract}\ v\ d\ e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Ann}\ a\ e) = \\
&\quad \mathit{Ann}\ a\ (\mathit{abstract}\ v\ d\ e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Add}\ e_1\ e_2) = \\
&\quad \mathit{Add}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Sub}\ e_1\ e_2) = \\
&\quad \mathit{Sub}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Mul}\ e_1\ e_2) = \\
&\quad \mathit{Mul}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{Eq}_{\mathit{Int}}\ e_1\ e_2) = \\
&\quad \mathit{Eq}_{\mathit{Int}}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad\ \ (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{LT}_{\mathit{Int}}\ e_1\ e_2) = \\
&\quad \mathit{LT}_{\mathit{Int}}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad\ \ (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{And}\ e_1\ e_2) = \\
&\quad \mathit{And}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\qquad (\mathit{abstract}\ v\ d\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abstract}\ v\ d\ (\mathit{If}\ e_1\ e_2\ e_3) = \\
&\quad \mathit{If}\ (\mathit{abstract}\ v\ d\ e_1) \\
&\quad\quad\ (\mathit{abstract}\ v\ d\ e_2) \\
&\quad\quad\ (\mathit{abstract}\ v\ d\ e_3)
\end{aligned}
$$

Module Feldspar.BackEnd.AST

This module contains the declaration of the AST of the low-level language (C).

```
module Feldspar.BackEnd.AST where
import qualified Prelude
import Prelude (Int,String)
import Feldspar.Types


  -- variables
type Var = (String,Types)


  -- C function
data Func ann =
  Func String [Var] [Stmt ann]


  -- C statement
data Stmt ann =
  If_C (Exp_C ann) [Stmt ann] [Stmt ann]
   | Assign String (Exp_C ann)
   | Declare Types String

  | Ann_Stmt ann (Stmt ann)


  -- C expressions
data Exp_C ann =
  Var_C String
   | Num Int
   | Infix (Exp_C ann) String (Exp_C ann)
   | Unary String (Exp_C ann)

  | Ann_ExpC ann (Exp_C ann)
```

Module Feldspar.BackEnd.Pretty

This module contains the code for pretty-printing the low-level AST. It uses John Hughes's and Simon Peyton Jones's Pretty Printer Combinators [Hug95].

```
{-# LANGUAGE FlexibleInstances #-}
module Feldspar.BackEnd.Pretty where
import qualified Prelude
import Prelude (($),map,foldl1)
import Text.PrettyPrint (Doc,text,int,parens,semi,space
  ,comma,lbrace,rbrace,vcat,nest
  ,($+$),($$),(<>),(<+>))
import qualified Data.List
import Feldspar.BackEnd.AST
import Feldspar.Types


class Pretty a where
  pretty :: a → Doc


instance Pretty ann ⇒
  Pretty (Exp_C ann) where
  pretty (Var_C x) = text x
  pretty (Num i) = int i
  pretty (Infix e1 op e2) = parens (pretty e1
                            <+> text op
                            <+> pretty e2)
  pretty (Unary op e) = parens (text op
                        <+> pretty e)


pretty (Ann_ExpC ann e) = text "/*"
                          <+> (pretty ann) <+>
                          text "*/"
                          <+> pretty e


instance Pretty ann ⇒
  Pretty (Stmt ann) where
```

```haskell
pretty (IfC e1 e2 e3) = text "if"
  <+> parens (pretty e1)
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty e2))
  $+$ rbrace
  $+$ text "else"
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty e3))
  $+$ rbrace

pretty (Assign v e) = text v <+> text "="
                      <+> pretty e <> semi

pretty (Declare t v) = pretty t <+> text v <> semi

pretty (AnnStmt ann st) = text "/*"
                          <+> (pretty ann) <+>
                          text "*/"
                          $$ pretty st

instance Pretty ann =>
  Pretty (Func ann) where

pretty (Func name vs body) =
  text "#include \"feldspar.h\""
  $+$ text "void" <+> text name
  <+> parens (commaCat (map pretty vs))
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty body))
  $+$ rbrace

instance Pretty Var where
  pretty (v,t) = pretty t <+> text v

instance Pretty Types where
  pretty Int32 = text "int32_t"
  pretty Bool = text "uint32_t"

commaCat :: [Doc] -> Doc
commaCat ds = foldl1 (<>) $
  Data.List.intersperse (comma <> space) ds
```

Module Feldspar.BackEnd.Derivings

In this module, the type classes *Eq*,*Functor*,*Foldable* and *Traversable* is derived for the AST of the low-level language.

```
  -- the code is omitted
```

Module Feldspar.BackEnd.BXPretty

This module contains the code needed to bidirectionalize the pretty-printing transformation.

```
{-# LANGUAGE Rank2Types #-}
module Feldspar.BackEnd.BXPretty where
import qualified Prelude
import Prelude (Eq (..),Show (..),(.),Int,id,String
  ,Bool (..),Functor (..),Read (..),Monad (..),Maybe (..)
  ,Either (..),map,filter,($),fst,¬,splitAt,read
  ,tail,(+),length,(∧))
```

```
import Text.PrettyPrint (Doc,int,text)
import Control.Monad (unless)
import Data.List (isPrefixOf,stripPrefix)
import Data.Foldable (toList)
import Data.Traversable (Traversable)
import Data.Function (on)
```

```
import BX (fromJust,fromList,index,assoc,validAssoc
  ,union,lookupAll)
import Feldspar.BackEnd.Pretty (Pretty (..))
```

```
  -- lexical tokens
data Token =
     Ann String
       -- the annotations (comments)
   | Etc String
       -- anything except annotations
  deriving Show
```

```haskell
  -- tokens are compared ignoring space
  -- and new-line characters
instance Eq Token where
  (Ann s) == (Ann s′) = ((==) ‘on‘
    (filter (λx → (x/ = '\n') ∧
      (x/ = ' ')))) s s′
  (Etc s) == (Etc s′) = ((==) ‘on‘
    (filter (λx → (x/ = '\n') ∧
      (x/ = ' ')))) s s′
  _ == _ = False
```

```haskell
  -- checking if a token is an annotation
isAnn :: Token → Bool
isAnn (Ann _) = True
isAnn _ = False
```

```haskell
  -- lexical tokenizer
tokenize :: String → Maybe [Token]
tokenize [] = Just []
tokenize ('/' : '*' : ' ' : xs) = do
  (before,after) ← splitBy " */" xs
  ts             ← tokenize after
  return $ (Ann before) : ts
tokenize (x : xs) = do
  ts ← tokenize xs
  return $ case ts of
    []            → Etc [x]     : ts
    (Ann _) : _   → Etc [x]     : ts
    (Etc y) : ts′ → Etc (x : y) : ts′
```

```haskell
  -- finding index of a string inside another string
infixAt :: Eq a ⇒ [a] → [a] → Maybe Int
infixAt needle haystack = infixAt′ 0 needle haystack
  where
    infixAt′ _ _ []          = Nothing
    infixAt′ i n hs
       | n ‘isPrefixOf‘ hs = Just i
       | True              = infixAt′ (i + 1) n (tail hs)
```

```
  -- spliting a string by the given key string
splitBy :: Eq a ⇒ [a] → [a] → Maybe ([a],[a])
splitBy infx s = do
  i ← infixAt infx s
  let (before,wafter) = splitAt i s
  after ← stripPrefix infx wafter
  return (before,after)
```

```
  -- The format of the output string of
  -- pretty printing us to extract the annotations
  -- by 1.tokenizing the string 2.extracting the
  -- the comments 3.parsing the strings to the
  -- actual annotation values, hence the Read
  -- constraint
toList_Doc :: ∀ a.Read a ⇒ String → [a]
toList_Doc d = [read s |
  Ann s ← fromJust $
  tokenize d]
```

```
  -- the shape of two output strings are
  -- compared by ignoring the values in the
  -- comments
eqShape_Doc :: String → String → Bool
eqShape_Doc = (==) `on`
  (fmap (filter (¬.isAnn))
  .tokenize)
```

```
-- since pretty printing uses type classes for
-- overloading, we are no longer able to use
-- our generic function (bff); we have to change
-- the code slightly (as highlighted)
bx_PP :: ∀ k.(Traversable k,Pretty (k Doc)) ⇒
  (∀ t.Pretty t ⇒
    k t → String) →
  (∀ a.(Read a,Eq a,Pretty a) ⇒
    k a → String →
    Either String (k a))
bx_PP get s v = do
  let sL = toList s
  let vL = toList_Doc v
  let get_By_L :: ∀ a.(Read a,Pretty a) ⇒
    [a] → [a]
    get_By_L x = highlight toList_Doc $
      get (fromList s x)
  unless (highlight eqShape_Doc (get s) v)
    $ Left "Modified view of wrong shape!"
  sL' ← bff_Pretty get_By_L sL vL
  return $ fromList s sL'
```

```
  -- the version of bff working with lists and
  -- pretty printing constraint; it does not
  -- check for validity of the mappings since
  -- the type Doc is abstract and the exposed
  -- observer functions by the module, namely
  -- the functions show and render are not
  -- used in our pretty printer
bff_Pretty :: (∀ a.(Read a,Pretty a) ⇒
      [a] → [a]) →
  (∀ a.(Eq a,Pretty a) ⇒
      [a] → [a] → Either String [a])
bff_Pretty get s v = do
    -- Step 1
  let ms = index s
    -- Step 2
  let is  = fst `map` ms
  let iv  = get is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms' = union mv ms
    -- Step 5.1
    -- check is removed
    -- Step 6
  return $ lookupAll is ms'
```

```
  -- the put (backward) function that
  -- bidirectionalizes the pretty printer
putPretty :: ∀ k a.
  (Eq a,Read a,Traversable k
  ,Pretty (k Doc),Pretty a) ⇒
  k a → String → Either String (k a)
putPretty = bx_PP (show.pretty.(fmap pretty))
```

```
instance Pretty Doc where
  pretty = id
instance Pretty Int  where
  pretty = int
instance Pretty Bool where
  pretty = text.show
```

Module Examples.TestFeldspar

This module contains an example program written in the high-level language.

```
openBrace −# OPTIONS_GHC − F − pgmF qapp #− closeBrace
```

```
module Examples.TestFeldspar where
import qualified Prelude
import Feldspar
import Feldspar.Compiler
```

```
inc :: Data Int32 → Data Int32
inc x = x + 1
```

```
dec :: Data Int32 → Data Int32
dec x = x − 1
```

```
incAbs :: Data Int32 → Data Int32
incAbs a = condition (a < 0) (dec a) (inc a)
```

```
cCode :: IO ()
cCode = icompile incAbs
```

C Code Examples.TestFeldspar

Listing 1: Pico-Feldspar/Examples/TestFeldspar.c

```c
#include "feldspar.h"
void test (int32_t v0, int32_t *out)
{
  /* False */
  int32_t v1;
  /* False */
  if (/* False */ (/* False */ v0 < /* False */ 0))
  {
    v1 = /* False */ (/* False */ v0 - /* False */ 1);
  }
  else
  {
    v1 = /* True */ (/* False */ v0 + /* False */ 1);
  }
  *out = /* False */ v1;
}
```

Module Examples.BXTestFeldspar

```haskell
module Example.BXTestFeldspar where
import Feldspar.AnnotationUtils (markAllF,markAll)
import Feldspar.BX              (putAnn)
import Examples.TestFeldspar (incAbs)
import Feldspar.Compiler.Compiler (scompile)


  -- the location of the generated C code
c :: String
c = "Examples/TestFeldspar.c"


  -- forward transformation from EDSL to C
forward :: IO ()
forward = writeFile c
  (scompile ((markAllF markAll)
    incAbs))
```

```
  -- backward transformation from C to src-loc
backward :: IO ()
backward = do
  cSrc ← readFile c
  let r = putAnn False (markAllF markAll)
    incAbs cSrc
  case r of
    Right locs → putStrLn $ show locs
    Left er → putStrLn er
```

Bibliography

[Hug95] John Hughes. The design of a pretty-printing library. *Advanced Functional Programming*, pages 53–96, 1995.