Use *deep embeddings* so that you can define multiple interpretations, and in particular simulate all your programs in different contexts.

Keep in mind that you can use both *initial* and *final* style – whatever works easier for you.

Define *meaningful application-specific interfaces* that express what you are actually doing.

Keep simulation in mind when designing your interfaces. Do not expose *unnecessary implementation details*. 
Flavours of Haskell EDSLs
Hutton’s Razor\(^1\)

A language with just two operations:

```
lit :: Integer -> Expr  -- integer literal
add :: Expr -> Expr -> Expr  -- addition
```

\(^1\)named after Graham Hutton
Hutton’s Razor\(^1\)

A language with just two operations:

\[
\begin{align*}
\text{lit} & : \text{Integer} \rightarrow \text{Expr} \quad \text{-- integer literal} \\
\text{add} & : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \quad \text{-- addition}
\end{align*}
\]

One of the most simple EDSLs one can define.

\(^{1}\text{named after Graham Hutton}\)
A language with just two operations:

\[
\begin{align*}
\text{lit} & : \text{Integer} \rightarrow \text{Expr} \quad \text{-- integer literal} \\
\text{add} & : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \quad \text{-- addition}
\end{align*}
\]

One of the most simple EDSLs one can define.

Example program:

\[
\begin{align*}
\text{term} & : \text{Expr} \\
\text{term} & = \text{lit} 1 \ \text{`add`} \ (\text{lit} 3 \ \text{`add`} \ \text{lit} 4)
\end{align*}
\]

\(^1\text{named after Graham Hutton}\)
The embedded program directly denotes its interpretation:

```haskell
type Expr = Integer
lit = id
add = (+)
```
The embedded program directly denotes its interpretation:

```
type Expr = Integer
lit = id
add = (+)
```

```
GHCi> term
8
```
Shallow embedding – evaluation

Pro:

▶ Very direct and simple.
▶ Usually quite performant.
▶ Easy to add new language constructs.

Con:

▶ Tied to a single interpretation / semantics.
▶ Interface of the EDSL is implicit.
▶ No analysis of the program possible.
▶ No proper abstraction.
The embedded program represents itself (and thereby all possible interpretations):

```haskell
data Expr =
    Lit Integer
  | Add Expr Expr

lit = Lit
add = Add
```

Well-Typed
Interpretation: evaluation

\[
\text{eval} :: \text{Expr} \rightarrow \text{Integer} \\
\text{eval} (\text{Lit } i) = i \\
\text{eval} (\text{Add } e_1 e_2) = \text{eval } e_1 + \text{eval } e_2
\]
Interpretation: evaluation

eval :: Expr -> Integer
eval (Lit i) = i
eval (Add e1 e2) = eval e1 + eval e2

GHCi> eval term
8
text :: Expr -> String

text (Lit i)  = show i

text (Add e1 e2) =
    "(" <> text e1 <> "+" <> text e2 <> ")"
text :: Expr -> String

\[
\begin{align*}
text \ (\text{Lit } i) &= \text{show } i \\
text \ (\text{Add } e1 \ e2) &= \\
&= "(" \ <> \ \text{text } e1 \ <> \ "+" \ <> \ \text{text } e2 \ <> \ ")"
\end{align*}
\]

GHCi> text \ term
"(1+(3+4))"
**Interpretation: instructions for a stack machine**

```haskell
data Instr =
  PUSH Integer
| ADD

compile :: Expr -> [Instr]
compile (Lit i)   = [PUSH i]
compile (Add e1 e2) =
  compile e1 ++ compile e2 ++ [ADD]
```

Well-Typed
Interpretation: instructions for a stack machine

```haskell
data Instr =
  PUSH Integer
| ADD

compile :: Expr -> [Instr]
compile (Lit i) = [PUSH i]
compile (Add e1 e2) =
  compile e1 ++ compile e2 ++ [ADD]

GHCi> compile term
[PUSH 1, PUSH 3, PUSH 4, ADD, ADD]
```

Well-Typed
Deep embedding – evaluation

Pro:

- Easy to define several interpretations.
- Easy to perform analysis and transformations of the program.
- Interface is explicit (via constructors of datatype).
- Hard to refer to a particular abstraction.

Con:

- Sometimes trickier in terms of performance (e.g. sharing).
- Harder to add new language constructs.
Using a type class instead

```haskell
class IsExpr e where
  lit :: Integer -> e
  add :: e -> e -> e
```

Using a type class instead

```haskell
class IsExpr e where
    lit :: Integer -> e
    add :: e -> e -> e

term :: IsExpr e => e
term = lit 1 `add` (lit 3 `add` lit 4)
```
Using a type class instead

```haskell
class IsExpr e where
    lit  :: Integer -> e
    add  :: e -> e -> e

term  :: IsExpr e -> e
term = lit 1 `add` (lit 3 `add` lit 4)
```

Slightly different interface:

```haskell
add  :: Expr -> Expr -> Expr
add  :: IsExpr e => e -> e -> e -> e
```
instance IsExpr Integer where

  lit :: Integer -> Integer
  lit = id

  add :: Integer -> Integer -> Integer
  add = (+)
instance IsExpr Integer where

  lit :: Integer -> Integer
  lit = id

  add :: Integer -> Integer -> Integer
  add = (+)

GHCi> term :: Integer
8
instance IsExpr String where

    lit :: Integer -> String
    lit i = show i

    add :: String -> String -> String
    add e1 e2 =
        "(" <> e1 <> "+" <> e2 <> ")"
**instance IsExpr String where**

lit :: Integer -> String
lit i = show i

add :: String -> String -> String
add e1 e2 =
  "(" <> e1 <> "+" <> e2 <> ")"

Note: No recursive calls.
instance IsExpr String where

  lit :: Integer -> String
  lit i = show i

  add :: String -> String -> String
  add e1 e2 =
    "(" <> e1 <> "+" <> e2 <> ")"

Note: No recursive calls.

GHCi> term :: Eval
  "(1+(3+4))"
instance IsExpr [Instr] where

  lit :: Integer -> [Instr]
  lit i = [PUSH i]

  add :: [Instr] -> [Instr] -> [Instr]
  add e1 e2 =
    e1 ++ e2 ++ [ADD]

GHCi> term :: [Instr]
  [PUSH 1, PUSH 3, PUSH 4, ADD, ADD]
Probably better: use a newtype

```haskell
newtype Eval = EvalC {unEval :: Integer}
deriving Show

instance IsExpr Eval where
    lit :: Integer -> Eval
    lit = coerce

    add :: Eval -> Eval -> Eval
    add = coerce ((+) :: Integer -> Integer -> Integer)
```

```
GHCi> term :: Eval
EvalC {unEval = 8}
```
Comparison

“Initial” style:

```haskell
data Expr =
    Lit Integer
  | Add Expr Expr
```

“Final” style:

```haskell
class IsExpr e where
    lit :: Integer -> e
    add :: e -> e -> e
```

Both are deep embeddings, with slightly different advantages and disadvantages.
instance IsExpr Expr where
  lit = Lit
  add = Add
from :: IsExpr e => Expr -> e
from (Lit i) = lit i
from (Add e1 e2) = add (from e1) (from e2)
We’ll focus on deep embeddings, because we want multiple interpretations of our programs, in particular:

▶ “real-world” execution,
▶ simulated execution.

We’ll still consider both initial and final style.
Adding effects
New interface

```haskell
data Var
data Expr
data Imp a
instance Monad Imp
new :: Imp Var
set :: Var -> Expr -> Imp ()
say :: Var -> Imp ()
var :: Var -> Expr
lit :: Integer -> Expr -- as before
add :: Expr -> Expr -> Expr -- as before
```
Example program

fib = do
  x <- new
  y <- new
  z <- new
  set x (lit 1)
  set y (lit 1)
  forever $ do
    say x
    set z (var x)
    set x (var y)
    set y (add (var z) (var y))
A deep embedding?

data Expr =
    Lit Integer
  | Add Expr Expr
  | Var Var

data Imp :: * -> * where
    New     :: Imp Var
    Set     :: Var -> Expr -> Imp ()
    Say     :: Var -> Imp ()

data Var
A deep embedding?

```haskell
data Expr =
    Lit Integer |
    Add Expr Expr |
    Var Var

data Imp :: * -> * where
    New :: Imp Var
    Set :: Var -> Expr -> Imp ()
    Say :: Var -> Imp ()

data Var
```

Two problems:

- What about `instance Monad Imp`?
- What about `Var`?
A deep embedding?

```
data Expr =
    Lit Integer
  | Add Expr Expr
  | Var Var

data Imp :: * -> * where
    New :: Imp Var
    Set :: Var -> Expr -> Imp ()
    Say :: Var -> Imp ()
    Return_{Imp} :: a -> Imp a
    Bind_{Imp} :: Imp a -> (a -> Imp b) -> Imp b

data Var
```
Instances

```
instance Monad Imp where
    return = Return\textsubscript{Imp}
    (>>=) = Bind\textsubscript{Imp}

instance Applicative Imp where
    pure = return
    (\langle\star\rangle) = ap

instance Functor Imp where
    fmap = liftM
```
Instances

```haskell
instance Monad Imp where
  return = Return_{Imp}
  (>>=) = Bind_{Imp}

instance Applicative Imp where
  pure = return
  (<*>) = ap

instance Functor Imp where
  fmap = liftM
```

Laws are not fulfilled on the syntactic level, therefore there is a proof obligation for each interpretation.

Or switch to a proper **free monad**.
Attempting an interpretation

Idea:

- Interpret \texttt{Imp} programs as \texttt{IO} actions.
- Represent variables as \texttt{IORef}s.

However, this immediately fails:

\begin{verbatim}
exec IO :: Imp a -> IO a
exec IO New = newIORef 0 -- :: IO (IORef Integer), not IO Var
\end{verbatim}

Because:

\texttt{New :: Imp Var -> Imp a}  

We must have the freedom to choose what \texttt{Var} is interpreted as.
Attempting an interpretation

Idea:

- Interpret $\text{Imp}$ programs as $\text{IO}$ actions.
- Represent variables as $\text{IORef}$s.

However, this immediately fails:

$$\text{exec}_{\text{IO}} :: \text{Imp } a \rightarrow \text{IO } a$$
$$\text{exec}_{\text{IO}} \text{ New} = \text{newIORef } 0 \quad --::\quad \text{IO } (\text{IORef } \text{Integer}), \not\text{IO Var}$$

Because:

$$\text{New} :: \text{Imp } \text{Var } \rightarrow \text{Imp } a$$
Attempting an interpretation

Idea:

- Interpret \texttt{Imp} programs as \texttt{IO} actions.
- Represent variables as \texttt{IORef} s.

However, this immediately fails:

```
exec_{IO} :: \texttt{Imp} a \rightarrow \texttt{IO} a
exec_{IO} \texttt{New} = \texttt{newIORef} 0 \quad --:: \texttt{IO (IORef Integer)\texttt{,notIO Var}}
```

Because:

```
\texttt{New} :: \texttt{Imp Var} \rightarrow \texttt{Imp a}
```

We must have the freedom to choose what \texttt{Var} is interpreted as.
Abstracting from \textbf{Var}

\begin{verbatim}
\textbf{data} Expr \texttt{var} =
\hspace{1cm} \text{Lit Integer}
\hspace{1cm} | \text{Add} (Expr \texttt{var}) (Expr \texttt{var})
\hspace{1cm} | \text{Var} \texttt{var}

\textbf{data} Imp :: \* \to \* \to \* \textbf{where}
\hspace{1cm} \text{New} :: \text{Imp} \texttt{var} \texttt{var}
\hspace{1cm} \text{Set} :: \texttt{var} \to \text{Expr} \texttt{var} \to \text{Imp} \texttt{var} ()
\hspace{1cm} \text{Say} :: \texttt{var} \to \text{Imp} \texttt{var} ()
\hspace{1cm} \text{Return}_{\text{Imp}} :: \texttt{a} \to \text{Imp} \texttt{var} \texttt{a}
\hspace{1cm} \text{Bind}_{\text{Imp}} :: \text{Imp} \texttt{var} \texttt{a} \to (\texttt{a} \to \text{Imp} \texttt{var} \texttt{b}) \to \text{Imp} \texttt{var} \texttt{b}
\end{verbatim}

Well-Typed
Interpretation: execution

```
exec IO :: Imp (IORef Integer) a -> IO a
exec IO New = newIORef 0
exec IO (Set v e) = do
  x <- eval IO e
  writeIORef v x
exec IO (Say v) = do
  x <- readIORef v
  print x
exec IO (Return Imp x) = return x
exec IO (Bind Imp m k) = exec IO m >>= exec IO . k
```

eval IO :: Expr (IORef Integer) -> IO Integer
eval IO (Lit i) = return i
eval IO (Add e1 e2) = liftM2 (+) (eval IO e1) (eval IO e2)
eval IO (Var v) = readIORef v
How to simulate without IO?

```haskell
newtype Counter = Counter {getCounter :: Integer}
  deriving (Show, Num, Eq, Ord)

data Sim a where
  Fresh :: Sim Counter
  Insert :: Counter -> Integer -> Sim ()
  Lookup :: Counter -> Sim Integer
  Message :: String -> Sim ()
  ReturnSim :: a -> Sim a
  BindSim :: Sim a -> (a -> Sim b) -> Sim b

instance Monad Sim where
  return = ReturnSim
  (>>>=) = BindSim
```

This is just Phase One, but Sim is quite a bit more low-level than Imp.

Well-Typed
How to simulate without IO?

```haskell
newtype Counter = Counter {getCounter :: Integer}
  deriving (Show, Num, Eq, Ord)

data Sim a where
  Fresh     :: Sim Counter
  Insert    :: Counter -> Integer -> Sim ()
  Lookup    :: Counter -> Sim Integer
  Message   :: String -> Sim ()
  ReturnSim :: a -> Sim a
  BindSim   :: Sim a -> (a -> Sim b) -> Sim b

instance Monad Sim where
  return = ReturnSim
  (>>>=)  = BindSim
```

This is just Phase One, but Sim is quite a bit more low-level than Imp.
Interpretation: simulation of programs

\[
\text{exec}_\text{Sim} :: \text{Imp} \text{ Counter} \, a \rightarrow \text{Sim} \, a
\]

\[
\text{exec}_\text{Sim} \, \text{New} = \text{do}
\]
\[
\quad v \leftarrow \text{Fresh}
\]
\[
\quad \text{Insert} \, v \, 0
\]
\[
\quad \text{return} \, v
\]

\[
\text{exec}_\text{Sim} \, (\text{Set} \, v \, e) = \text{do}
\]
\[
\quad x \leftarrow \text{eval}_\text{Sim} \, e
\]
\[
\quad \text{Insert} \, v \, x
\]

\[
\text{exec}_\text{Sim} \, (\text{Say} \, v) = \text{do}
\]
\[
\quad x \leftarrow \text{Lookup} \, v
\]
\[
\quad \text{Message} \, (\text{show} \, x)
\]

\[
\text{exec}_\text{Sim} \, (\text{Return}_\text{Imp} \, x) = \text{return} \, x
\]

\[
\text{exec}_\text{Sim} \, (\text{Bind}_\text{Imp} \, m \, k) = \text{exec}_\text{Sim} \, m \gg= \text{exec}_\text{Sim} \, . \, k
\]
Interpretation: simulation of expressions

\[
\begin{align*}
\text{eval}_\text{Sim} &:: \text{Expr} \text{ Counter} \rightarrow \text{Sim} \text{ Integer} \\
\text{eval}_\text{Sim} (\text{Lit} \ i) &= \text{return} \ i \\
\text{eval}_\text{Sim} (\text{Add} \ e_1 \ e_2) &= \text{liftM2} (\text{+}) (\text{eval}_\text{Sim} \ e_1) (\text{eval}_\text{Sim} \ e_2) \\
\text{eval}_\text{Sim} (\text{Var} \ v) &= \text{Lookup} \ v
\end{align*}
\]
Phase Two

One option is:

```
type SimResult = Stream (Of String) (State SimState)
```
One option is:

```haskell
type SimResult = Stream (Of String) (State SimState)
```

A `Stream` (from the streaming package) is a way to interleave items and effects:

```haskell
data Stream f m r =
  Step  !(f (Stream f m r))
  | Effect (m (Stream f m r))
  | Return r

data Of a b = !a :> b

instance (Functor f, Monad m) => Monad (Stream f m)
instance Functor f => MonadTrans (Stream f)
...
```
One option is:

```haskell
type SimResult = Stream (Of String) (State SimState)
```

```haskell
data SimState = SimState
  { _ctr :: Counter,
    _env :: Map Counter Integer
  }
```

```haskell
ctr :: Lens' SimState Counter
env :: Lens' SimState (Map Counter Integer)
```
Phase One to Phase Two

```haskell
runSim :: Sim a -> SimResult a
runSim Fresh = lift $ do
  v <- use ctr
  ctr += 1
  env %= insert v 0
  return v
runSim (Insert v x) = lift $ env %= insert v x
runSim (Lookup v) = lift $ (! v) <$> use env
runSim (Message m) = yield m
runSim (ReturnSim x) = return x
runSim (BindSim m k) = runSim m >>= runSim . k
```
testFib :: [String]
testFib =
    evalState
        (S.toList_ (S.take 5 (runSim (execSim fib))))
        (SimState {_ctr = 0, _env = empty})

GHCi> testFib
["1", "1", "2", "3", "5"]
\textbf{class}
\[
(\text{IsExpr } e \; v, \text{Monad } i) \Rightarrow \text{IsImp } i \; e \; v | i \to e \; v \\
\text{where}
\]
\text{new} :: i \; v \\
\text{set} :: v \to e \to i () \\
\text{say} :: v \to i ()

\textbf{class} \text{IsExpr } e \; v | e \to v \text{ where}
\textbf{class} \text{IsExpr } e \; v | e \to v \text{ where}

\text{var} :: v \to e \\
\text{lit} :: \text{Integer} \to e \\
\text{add} :: e \to e \to e
A few observations
Abstraction over `Var` was important

- Fixing such a resource type to a concrete choice can easily limit us to one (or just a few interpretations).
- Be careful with anything like handles, database connections, variables, stores, or any abstract types where the interface itself ties you into specific monads.
Avoid direct use of `IO` or `MonadIO` in your domain-specific code at all costs.

Try to capture the operations you actually need in the interface directly. It has the added side effect that it becomes clearer what exactly the code is allowed and supposed to do.
Few large interfaces

- We only used actual monad transformers at the lowest level, when implementing Sim.
- How exactly we implemented Sim is irrelevant to the rest of the program (monad transformers, extensible effects, monolithic monad, ...).
- Too much granularity in interfaces often makes things more complicated rather than less. Only split if there are real use cases where you need one without the other, or where the interfaces really are at different levels.
Effect-free interfaces are still best

- Nothing beats the simplicity of the first scenario.
- Even if interpretation require effects, the DSL not necessarily does.
- Write code as pure functions if you can.