





DEVELOPING DSLS

A LOOK AT THREE PRACTICAL STRATEGIES
WITH REAL-WORLD EXAMPLES

Ziyang Liu

Three Common Themes

	Examples	Examples 
 Standalone External	SQL, JSON, Yaml, Verilog, VHDL, Jacinda	Aiken, Helois, Untyped Plutus Core
 Embedded Internal	Copilot, Protocols, Hardcaml, Feldspar, YuIDSL	Plutarch, plu-ts
 Subsets of GPLs	Categorifier, QFeldspar	Plutus Tx, OpShin, Scalus

1. Standalone | External DSLs



- ▶ Highly customizable for the domain, including syntax and functionalities
- ▶ No baggage to carry from a host language, which is often designed for different purposes

1. Standalone | External DSLs



- ▶ Requires building from scratch: compiler components, tooling, library ecosystem, documentation, and more
- ▶ Learning a new DSL can be harder than anticipated – even with a fluent syntax
 - ▶ Scarcity of learning resources
 - ▶ Past experiences may not be useful
 - ▶ Knowledge learned may not transfer well
 - ▶ Additional languages, additional complexity



1. Standalone | External DSLs

- ▶ Great choices for low level languages and compilation targets
 - ▶ Small, custom languages are easier to target
 - ▶ Flexibility on runtime
 - ▶ Easier to reason about and formalize
 - ▶ Avoid some work required for building surface languages
- ▶ Exercise caution when using for surface languages

BEST OF BOTH WORLDS

standalone DSLs for compilation targets
reusing GPLs for surface languages

Writing basic programs in a popular GPL is easier than you might think



Even for Haskell!



- ▶ Abundance of learning resources - books, AI, StackOverflow, Reddit ...
- ▶ Library ecosystem
- ▶ Transferable knowledge

Standalone DSL Examples

Untyped Plutus Core (UPLC)

- ▶ Low level smart contract language for Cardano
- ▶ Based on untyped lambda calculus

UPLC: find the first integer in the given list smaller than 10

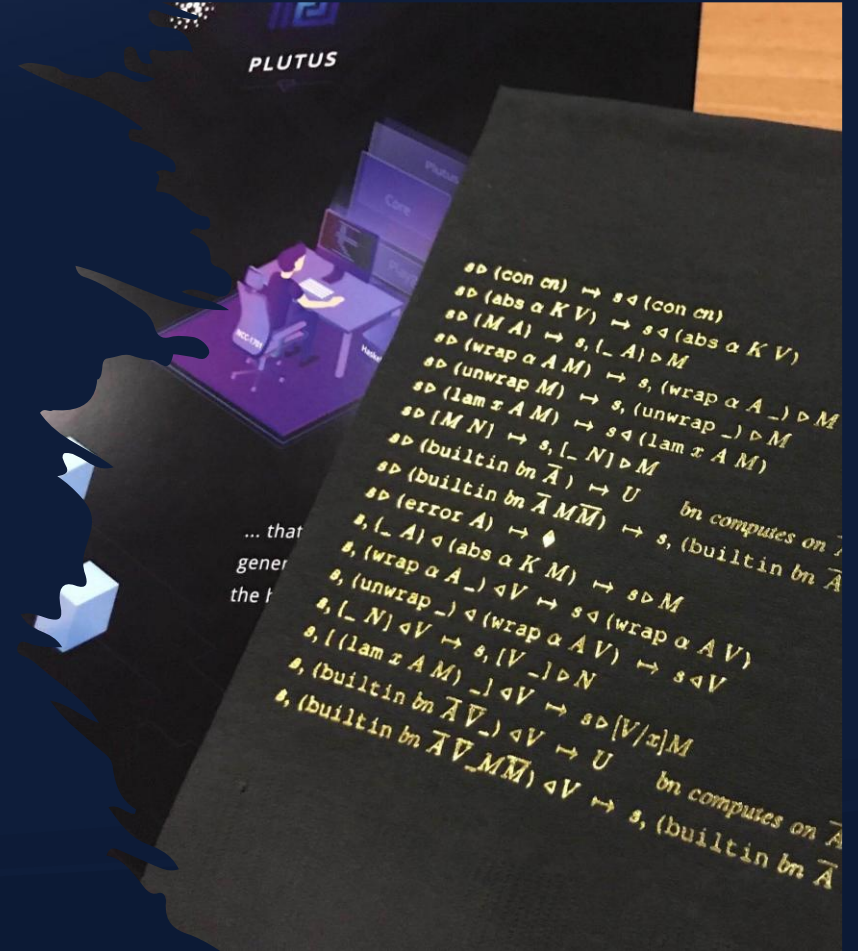
```
(\s -> s s)
(\s ds ->
  force
  (case ds
    [ (delay (constr 1 []))
    , (\x xs ->
      delay
      (force
        (force
          (force ifThenElse
            (lessThanInteger x 10)
            (delay (delay (s s xs)))
            (delay (delay (constr 0 [x]))))
          )
        )
      )
    ]
  )
)
```

Standalone DSL Examples

Untyped Plutus Core (UPLC)

Why UPLC?

- ▶ Tailored for the Cardano smart contract domain
- ▶ Small — evaluator fits on a napkin
- ▶ Stable
- ▶ Well-researched



*“How do you make sure that your language won't be obsolete in 40 years?
Use one that was invented 40 years ago, and use one that was discovered twice.”*

— PHILIP WADLER —

Standalone DSL Examples

Configuration Languages (JSON, YAML etc.)

- ▶ Simple, readable standalone DSLs for configuration
- ▶ Well-suited to be compilation targets (just like UPLC)
 - ▶ Safer, less repetition

Standalone DSL Examples

Aiken

- ▶ A good example of a standalone DSL as a surface language
- ▶ Curly bracket syntax, familiar to a broad audience
- ▶ Tailor-made for Cardano smart contracts:
 - ▶ Keywords like “validator” and “spend”
 - ▶ Built-in mechanisms for tracing – “trace” and “?”
 - ▶ Hiding data constructors with the “opaque” keyword
 - ▶ Writing everything including tests in one module
 - ▶ Specific, tailored error messages

Aiken, a standalone DSL targeting UPLC

```
validator hello_world {
  spend(
    datum: Option<Datum>,
    redeemer: Redeemer,
    _own_ref: OutputReference,
    self: Transaction,
  ) {
    trace @"redeemer": string.from_bytearray(redeemer.msg)
    expect Some(Datum { owner }) = datum
    let must_say_hello = redeemer.msg == "Hello, World!"
    let must_be_signed = list.has(self.extra_signatories, owner)
    must_say_hello? && must_be_signed?
  }
}

test hello_world_example() {
  let datum =
    Datum { owner: #"0000000000000000000000000000000000000000000000000000000000000000" }
  let redeemer = Redeemer { msg: "Hello, World!" }
  let placeholder_utxo = OutputReference { transaction_id: "", output_index: 0 }

  hello_world.spend(
    Some(datum),
    redeemer,
    placeholder_utxo,
    Transaction { ..transaction.placeholder, extra_signatories: [datum.owner] },
  )
}
```

2. Embedded | Internal DSLs AST Manipulation Libraries



- ▶ Reuse of host language infrastructure
 - ▶ Very desirable if you are already using the host language
- ▶ Still customizable for your domain
 - ▶ Programming model and semantics can diverge from those of the host language's
- ▶ Easiest to develop, maintain and document, especially when embedded in a statically typed functional host language
 - ▶ Well-suited for an MVP
 - ▶ Embed your DSL in multiple host languages
- ▶ Metaprogramming is generally easier



2. Embedded | Internal DSLs

- ▶ AST construction and manipulation are exposed to users

IDEALLY: ($>$) :: Int \rightarrow Int \rightarrow Bool
REALITY: ($.>$) :: AST Int \rightarrow AST Int \rightarrow AST Bool

```
Plutarch: an eDSL targeting UPLC

fib :: Term s (PInteger --> PInteger)
fib = phoistAcyclic $
  pfix # $ plam $ \self n ->
    pif
      (n #== 0)
      0
      $ pif
        (n #== 1)
        1
        $ self # (n - 1) + self # (n - 2)
```

```
Copilot: an eDSL targeting C for stream processing

{-# LANGUAGE RebindableSyntax #-}

module Main where

import Language.Copilot
import Copilot.Compile.C99

-- A resettable counter
counter :: Stream Bool -> Stream Bool -> Stream Int32
counter inc reset = cnt
  where
    cnt = if reset then 0
          else if inc then z + 1
                else z
    z = [0] ++ cnt

-- Counter that resets when it reaches 256
bytecounter :: Stream Int32
bytecounter = counter true reset
  where
    reset = counter true false `mod` 256 == 0
```

“Additional efforts can hide more of these symptoms, but the cracks still show, and each new coping mechanism leads to increasingly mysterious type errors.”
 — CONAL ELLIOTT —

```
an eDSL targeting C for flight controller development

batteryCurrents :: Batteries (f Double)
batteryCurrents =
  bool 0 currentPerBattery <$> batteriesConnectedToBus
  where
    bool f t predicate =
      kIfThenElse predicate t f
    currentPerBattery :: f Double
    currentPerBattery =
      let numConnected :: f Word32
          numConnected =
            sum . fmap (bool 0 1) $ batteriesConnectedToBus
      in kIfThenElse
        (numConnected .> 0)
        (sum mcCurrents / kFromIntegral numConnected)
        0
```

2. Embedded | Internal DSLs




- ▶ Challenging to produce readable target code or develop debuggers
- ▶ The host language's library ecosystem provides limited value


All of these also applies to shallow embeddings,
which are equivalent to deep embeddings
(Fix vs. Mu)

3. Subsets of GPLs

Reusing Host Language ASTs



IDEALLY: (>) :: Int → Int → Bool 

REALITY: (>) :: Int → Int → Bool 

METAPROGRAMMING - QDSL

- ▶ Access to surface language ASTs (usually relatively large)
- ▶ Heavy syntactic noise
- ▶ Still not regular programming
- ▶ Host language libraries not useful

COMPILER PLUGINS – TRUE SUBSETS

- ▶ Access to both surface and lower-level ASTs
- ▶ Little to no syntactic noise
- ▶ Write plain vanilla simple programs
- ▶ Reuse host language libraries

THE SPECIFICS VARY DEPENDING ON THE HOST LANGUAGE

data **HsExpr** p

A Haskell expression.

Constructors

HsVar (XVar p) (LIIdP p)

HsUnboundVar (XUnboundVar p) RdrName

HsRecSel (XRecSel p) (FieldOcc p)

HsOverLabel (XOverLabel p) SourceText FastString

HsIPVar (XIPVar p) HsIPName

HsOverLit (XOverLitE p) (HsOverLit p)

HsLit (XLitE p) (HsLit p)

HsLam

(XLam p)

HsLamVariant

(MatchGroup p (LHsExpr p))

Tells whether this is for lambda

LamSingle: one match of

AnnKeywordId : AnnLa

Haskell's Surface AST



```
HsApp (XApp p) (LHsExpr p) (LHsExpr p)
HsAppType (XAppTypeE p) (LHsExpr p) (LHsWcType (NoGhcTc p))

OpApp (XOpApp p) (LHsExpr p) (LHsExpr p) (LHsExpr p)
NegApp (XNegApp p) (LHsExpr p) (SyntaxExpr p)

HsPar
  (XPar p)
  (LHsExpr p) Parenthesised expr; see Note [Parens in HsSyn]

SectionL (XSectionL p) (LHsExpr p) (LHsExpr p)
SectionR (XSectionR p) (LHsExpr p) (LHsExpr p)
ExplicitTuple (XExplicitTuple p) [HsTupArg p] Boxity

ExplicitSum (XExplicitSum p) ConTag SumWidth (LHsExpr p)

HsCase (XCase p) (LHsExpr p) (MatchGroup p (LHsExpr p))
```

```
HsIf (XIf p) (LHsExpr p) (LHsExpr p) (LHsExpr p)

HsMultiIf (XMultiIf p) [LGRHS p (LHsExpr p)]

HsLet (XLet p) (HsLocalBinds p) (LHsExpr p)

HsDo (XDo p) HsDoFlavour (XRec p [ExprLStmt p])

ExplicitList (XExplicitList p) [LHsExpr p]

RecordCon
  rcon_ext :: XRecordCon p
  rcon_con :: XRec p (ConLikeP p)
  rcon_flds :: HsRecordBinds p

RecordUpd
  rupd_ext :: XRecordUpd p
  rupd_expr :: LHsExpr p
  rupd_flds :: LHsRecUpdFields p
```

```
HsGetField
  gf_ext :: XGetField p
  gf_expr :: LHsExpr p
  gf_field :: XRec p (DotFieldOcc p)

HsProjection
  proj_ext :: XProjection p
  proj_flds :: NonEmpty (XRec p (DotFieldOcc p))

ExprWithTySig (XExprWithTySig p) (LHsExpr p) (LHsSigWcType (NoGhcTc p))

ArithSeq (XArithSeq p) (Maybe (SyntaxExpr p)) (ArithSeqInfo p)

HsTypedBracket (XTypedBracket p) (LHsExpr p)

HsUntypedBracket (XUntypedBracket p) (HsQuote p)
HsTypedSplice (XTypedSplice p) (LHsExpr p)

HsUntypedSplice (XUntypedSplice p) (HsUntypedSplice p)
HsProc (XProc p) (LPat p) (LHsCmdTop p)

HsStatic (XStatic p) (LHsExpr p)

HsPragE (XPragE p) (HsPragE p) (LHsExpr p)
HsEmbTy (XEmbTy p) (LHsWcType (NoGhcTc p))
XExpr !(XXExpr p)
```

Haskell's Core AST



```
data Expr b
```

This is the data type that represents GHCs core intermediate language.

Constructors

```
Var Id
Lit Literal
App (Expr b) (Arg b) | infixl 4 |
Lam b (Expr b)
Let (Bind b) (Expr b)
Case (Expr b) b Type [Alt b]
Cast (Expr b) CoercionR
Tick CoreTickish (Expr b)
Type Type
Coercion Coercion
```


QDSL in Haskell

Everything Old Is New Again: Quoted Domain-Specific Languages

Shayan Najd The University of Edinburgh sh.najd@ed.ac.uk	Sam Lindley The University of Edinburgh sam.lindley@ed.ac.uk	Josef Svenningsson Chalmers University of Technology josefs@chalmers.se	Philip Wadler The University of Edinburgh wadler@inf.ed.ac.uk
--	--	---	---

Abstract

We describe a new approach to implementing Domain-Specific Languages (DSLs), called Quoted DSLs (QDSLs), that is inspired by two old ideas: quasi-quotation, from McCarthy's Lisp of 1960, and the subformula principle of normal proofs, from Gentzen's natural deduction of 1935. QDSLs reuse facilities provided for the host language, since host and quoted terms share the same syntax, type system, and normalisation rules. QDSL terms are normalised to a canonical form, inspired by the subformula principle, which guarantees that one can use higher-order types in the source while guaranteeing first-order types in the target, and enables using types to guide fusion. We test our ideas by re-implementing Feldspar, which was originally implemented as an Embedded DSL (EDSL), as a QDSL; and we compare the QDSL and EDSL variants. The two variants produce identical code.

Categories and Subject Descriptors D.1.1 [Applicative (Functional) Programming]; D.3.1 [Formal Definitions and Theory]; D.3.2 [Language Classifications]: Applicative (functional) languages

Keywords domain-specific language, DSL, EDSL, QDSL, embedded language, quotation, normalisation, subformula principle

1. Introduction

Implementing domain-specific languages (DSLs) via quotation is one of the oldest ideas in computing, going back at least to McCarthy's Lisp, which was introduced in 1960 and had macros as early as 1963. Today, a more fashionable technique is Embedded DSL (EDSL), which reuses the facilities of the host language to

ensure that other DSLs might benefit from the same technique, particularly those that perform staged computation, where host code at generation-time computes target code to be executed at run-time. Generality starts at two. Here we test the conjecture of Cheney *et al.* (2013) by reimplementing the EDSL Feldspar (Axelsson *et al.* 2010) as a QDSL. We describe the key features of the design, and introduce a sharpened subformula principle. We compare QDSL and EDSL variants of Feldspar and assess the trade-offs between the two approaches.

QDSL terms are represented in a host language by terms in quotations (or more properly, quasi-quotations), where domain-specific constructs are represented as constants (free variables) in quotations. For instance, the following Haskell code defines a function that converts coloured images to greyscale, using QDSL Feldspar as discussed throughout the paper:

```
greyscale :: Qt (Img → Img)
greyscale = [|λimg →
  $mapImg
  (λr g b →
    let q = div ((30 × r) + (59 × g) + (11 × b)) 100
    in $mkPxl q q q img|]
```

We use a typed variant of Template Haskell (TH), an extension of GHC (Mainland 2012). Quotation is indicated by [|...|], anti-quotation by \$\$(...), and the quotation of a Haskell term of type *a* has type *Qt a*. The domain-specific constructs used in the code are addition, multiplication, and division. The anti-quotations `$mapImg` and `$mkPxl` denote splicing user-defined functions named `mapImg` and `mkPxl`, which themselves are defined as quoted terms: `mapImg` is a higher-order function that applies a

```
power' :: Int → Qt (Float → Maybe Float)
```

```
power' n =
```

```
  if n < 0 then
```

```
    [|λx → if x == 0 then Nothing
```

```
      else do y ← $$ (power' (-n)) x
```

```
        return (1 / y)|]
```

```
  else if n == 0 then
```

```
    [|λx → return 1|]
```

```
  else if even n then
```

```
    [|λx → do y ← $$ (power' (n div 2)) x
```

```
      return (y × y)|]
```

```
  else
```

```
    [|λx → do y ← $$ (power' (n - 1)) x
```

```
      return (x × y)|]
```

```
power'' :: Int → Qt (Float → Float)
```

```
power'' n =
```

```
  [|λx → maybe 0 (λy → y) ($$ (power' n) x)|]
```

Example of Subsets of GPLs Plinth (formerly Plutus Tx)

Plutus Tx, a subset of Haskell targeting UPLC

```
nqueens :: Integer -> Labeler -> [State]
nqueens n algorithm = (search algorithm (queens n))
{-# INLINABLE nqueens #-}

unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys =
  xs ++ foldl (flip (deleteBy eq)) (nubBy eq ys) xs
{-# INLINABLE unionBy #-}

data ConflictSet = Known [Var] | Unknown

knownConflict :: ConflictSet -> Bool
knownConflict (Known (a:as)) = True
knownConflict _               = False
{-# INLINABLE knownConflict #-}

checkComplete :: CSP -> State -> ConflictSet
checkComplete csp s =
  if complete csp s then Known [] else Unknown
{-# INLINABLE checkComplete #-}

search :: Labeler -> CSP -> [State]
search labeler csp =
  ( map fst
    . filter (knownSolution . snd)
    . leaves
    . prune (knownConflict . snd)
    . labeler csp
    . mkTree
  ) csp
{-# INLINABLE search #-}
```

Subsets of GPLs



- ▶ Limited applicability: host language paradigms, ASTs and compiler components may not be good fits for your domain
 - ▶ Such as when your domain requires an unconventional data or programming model
 - ▶ e.g., SQL
- ▶ Plugin support in compilers is often limited and unstable
- ▶ Host language ASTs and compiler components may not be a good fit

THANK YOU

1. Plinth (formerly Plutus Tx). <https://github.com/IntersectMBO/plutus>
2. Aiken. <https://aiken-lang.org/>
3. Categorifier. <https://github.com/con-kitty/categorifier>
4. Copilot. <https://github.com/Copilot-Language/copilot>
5. Feldspar. <https://github.com/Feldspar/feldspar-language>
6. Hardcaml. <https://github.com/janestreet/hardcaml>
7. Helios. <https://github.com/HeliosLang/compiler>
8. Jacinda. <https://vmchale.github.io/jacinda/>
9. OpShin. <https://github.com/OpShin/opshin>
10. Protocols. <https://github.com/edwinb/Protocols>
11. QFeldspar. <https://github.com/Feldspar/feldspar-language>
12. plu-ts. <https://github.com/HarmonicLabs/plu-ts>
13. Plutarch. <https://github.com/Plutonomicon/plutarch-plutus>
14. Scalus. <https://github.com/nau/scalus>
15. YulDSL. <https://github.com/yolc-dev/yul-dsl-monorepo>