Towards verification-friendly UPLC code generation

KPlutus Team
Runtime Verification Inc.

October 25, 2022

Abstract

KPlutus is ready for proving reachability properties over Untyped Plutus Core (UPLC) code. However, to prove properties of *PlutusTx contracts*, there are some requirements that must be fulfilled, regarding the *structure* of a given UPLC code. In this report we draft some of these requirements.

1 Introduction

KPlutus project is about analyzing Untyped Plutus Core (UPLC) programs using the K framework based on a K implementation of the CEK machine of UPLC. Untyped Plutus Core is a "dialect" of the lambda calculus with built-in datatypes and functions.

From the one hand, KPlutus is ready to run UPLC code and to prove reachability properties. From the other hand, in order to prove properties about PlutusTx contracts, there are some requirements that must be met. The remainder of this report discusses them after recalling PlutusTx compilation pipeline and introducing the uniformity problem of UPLC code generation from PlutusTx contracts.

2 The pipeline

In this project, contracts are written in PlutusTx, a Haskell plugin implemented in Template Haskell. Here is the pipeline:

1. GHC: Haskell → GHC Core

2. Plutus Tx compiler: GHC Core → Plutus IR

3. Plutus IR compiler: Plutus IR → Typed Plutus Core

4. Type eraser: Typed Plutus Core → Untyped Plutus Core

3 The problem: Uniform UPLC code generation

Different contracts may generate UPLC code structured in different ways. We have identified differences in the following elements:

- Choice of fixed point operator. Different choices for fix point operators can be done. (See Sec. 4.1.)
- Generation of meaningful identifiers. Onchain code uses De Bruijn which is not friendly to verification. (See Sec. 4.2.)

- Compilation of datatypes. Datatypes are compiled using the so-called Scott encoding. However, this falls prey to many optimizations and inlinings performed by PlutuxTx compiler and GHC. (See Sec. 4.2.2 and Sec. 4.3.1.)

4 Towards a uniform way

In the following sections we outline the UPLC code structure we expect from the compilation of a PlutusTx contract.

4.1 Fix point operator

The fix point operator we expect in the generated code is the following one, that we call *REC*.

$$REC \equiv \lambda f(\lambda s(ss))(\lambda s(\lambda x(f(ss))x))$$

In UPLC, this combinator is encoded as:

```
(lam f_0
     [ (lam s_0 [ s_0 s_0 ] )
       (lam s_0 (lam x_0 [ [ f_0 [ s_0 s_0 ] ] x_0 ] ) ) ] )
```

which is β-equivalent to the perhaps more standard Z-combinator used in strict functional languages,

$$Z \equiv \lambda f(\lambda s f(\lambda x(ssx)))(\lambda s f(\lambda x(ssx)))$$

and is encoded in UPLC as follows:

```
(lam f_0
     [ (lam s_0 [ f_0 (lam x_0 [ s_0 s_0 x_0 ]) ])
       (lam s_0 [ f_0 (lam x_0 [ s_0 s_0 x_0 ]) ]) ])
```

4.2 Meaningful identifiers

UPLC code runs onchain, therefore is must be as compact and run as efficiently as possible. To this end, one compilation choice is to generate programs using De Bruijn indices. [1]

4.2.1 De Bruijn indices

Each De Bruijn index is a natural number that represents an occurrence of a variable in a λ-term. It denotes the number of binders that are in scope between that occurrence and its corresponding binder.

For example:

- The term $\lambda x \lambda y x$, sometimes called the K combinator, is written as $\lambda\lambda 2$ with De Bruijn indices. The binder for the occurrence x is the second λ in scope.
- The term $\lambda x \lambda y \lambda z x z(yz)$ (the so-called S combinator), with De Bruijn indices, is $\lambda\lambda\lambda 31(21)$.

[1] DeBruijn is pronounced [dəˈbrœyn].

The check for α-equivalence of terms with De Bruijn indices is the same as that for syntactic equality. However, from the perspective of verification, we need proper symbols in UPLC output in order to perform semi-automatic verification. It makes it really hard to reflect back the code being generated to the source code using just the indices.

Depending on the version of Plutus, it may be necessary to define the following function, which is usually composed with `PlutusTx.getUplc`.

```
unDeBruijnProgram
     :: Program NamedDeBruijn uni fun ann -> QuoteT (Either FreeVariableError)
                                                    (Program Name uni fun ann)
unDeBruijnProgram (Program ann ver term) =
  Program ann ver Haskell.<$> unDeBruijnTerm term
```

4.2.2 Stopping optimizations in the PlutusTx compiler

The following arguments should be passed in the `ghc-options` section of your `<<contract>>.cabal`, where `<<contract>>` is the name of your project such as `native-tokens`, `stablecoin` or `djed`.

```
-fplugin-opt PlutusTx.Plugin:max-simplifier-iterations=0
-fforce-recomp
```

4.3 Bindings for code generation

4.3.1 The inlining problem

Currently, PlutusTx compiler achieves some sort of modular compilation by means of unfoldings. Unfoldings are the copies of functions that GHC uses to enable cross-module inlining. It's a way of getting the source of functions. Consequently, functions that are used transitively by Plutus Tx code must be marked as `INLINABLE`, which ensures that unfoldings are present.

Of course, inlining is a bad thing for verification for the same reason De Bruijn indices are: one looses symbols in the generated code that need to be there for semi-automated program verification to be possible.

4.3.2 The "inlinable-noinline pattern" solution

The solution we have at the moment for the inlining problem declares a wrapper binding, which is annotated as inlinable, whose single local variable binds to the binding we want to generate code for. And that single variable is annotated as `NOINLINE`.

A wrapper binding looks like the following:

```
{-# INLINABLE f #-}}
f :: A -> B
f a = ...

{-# INLINABLE f' #-}
f' : A -> B
f' a = f''
where f'' = f a
{-# NOINLINE f'' #-}
```

4.3.3 Policy code generation

Even though one can generate UPLC code for any element of a contract, we are interested in verifying *onchain* code, which essentially means the contract's policy, being it a validator or a minting one.

The pattern we are using at the moment has two bindings. The first binding holds the generated code, resulting from the application of `PlutusTx.compile` to the policy one wishes to generate code for. The second binding is a `String` resulting from the pretty-printing of the former.

4.3.3.1 Language extension and module importation In the module whose elements will be compiled to UPLC, it's necessary to add the language extension

```
{-# LANGUAGE DataKinds #-}
```

for Template Haskell to work properly, and the following module importations, for the code generation.

```
import UntypedPlutusCore
import PlutusTx
import PlutusCore.Quote
import PlutusCore.Pretty
import Prelude as Haskell
```

4.3.3.2 Validator case The pattern we describe here was obtained from the Stablecoin contract. A validator is understood as a function with the following codomain.

```
  Plutus.Script.Utils.V1.Typed.Scripts.Validators.UntypedValidator
```

Hence, the type of the binding for the code associated with a validator is

```
  PlutusTx.CompiledCode(
    Plutus.Script.Utils.V1.Typed.Scripts.Validators.UntypedValidator)
```

The `typedValidator` function is an example of it.

```
typedValidator :: Stablecoin ->
  Scripts.TypedValidator (StateMachine BankState Input)
typedValidator stablecoin =
    let val = $$(PlutusTx.compile [|| validator ||])
                  `PlutusTx.applyCode` PlutusTx.liftCode stablecoin
        validator d = SM.mkValidator (stablecoinStateMachine d)
        wrap = Scripts.mkUntypedValidator @BankState @Input
    in Scripts.mkTypedValidator
         @(StateMachine BankState Input) val $$(PlutusTx.compile [|| wrap ||])
```

The binding for the associated UPLC code is as follows:

```
typedValidator' ::
  PlutusTx.CompiledCode (Stablecoin -> Validators.UntypedValidator)
typedValidator' =
    let validator =
      Validators.mkUntypedValidator . SM.mkValidator . stablecoinStateMachine
    in $$(PlutusTx.compile [|| validator ||])
```

with the following `String` constants bound to the associated UPLC and PIR code (See Sec. 2):

```
uplcStableCoinPolicy :: Haskell.String
uplcStableCoinPolicy =
  either display displayPlcDebug $ runQuoteT $ unDeBruijnProgram $
    PlutusTx.getPlc typedValidator'

pirStableCoinPolicy = prettyClassicDebug $ PlutusTx.getPir typedValidator'
```

4.3.3.3 Minting case The pattern for the minting policy comes from the NFT contract, where `Scripts` denote `Ledger.Typed.Scripts`.

```
policy :: TxOutRef -> TokenName -> Scripts.MintingPolicy
policy oref tn = mkMintingPolicyScript $
    $$(PlutusTx.compile
       [|| \oref' tn' -> Scripts.wrapMintingPolicy $ mkPolicy oref' tn' ||])
    `PlutusTx.applyCode`
    PlutusTx.liftCode oref
    `PlutusTx.applyCode`
    PlutusTx.liftCode tn

compiledNFTPolicy :: (PlutusTx.CompiledCode
                        (TxOutRef -> TokenName ->
                         Scripts.WrappedMintingPolicyType))
compiledNFTPolicy = $$(PlutusTx.compile
  [|| \oref' tn' -> Scripts.wrapMintingPolicy $ mkPolicy oref' tn' ||])
```

Note that for the minting policy we needed to adjust the Haskell code that generates the UPLC code for the given policy, where `uplcDeBruijnProgram` is composed with `@(QuoteT (Either FreeVariableError))`. For PIR however the code pattern remains the same.

```
uplcNFTPolicy :: String
uplcNFTPolicy =
  either display displayPlcDebug $ runQuoteT $
  unDeBruijnProgram @(QuoteT (Either FreeVariableError)) $
    PlutusTx.getPlc compiledNFTPolicy

pirNFTPolicy :: String
pirNFTPolicy =
    PlutusCore.Pretty.prettyClassicDebug $ PlutusTx.getPir compiledNFTPolicy
```

4.4 Compilation of datatypes

Datatypes are compiled using the so-called Scott encoding, which, for each datatype, generates constructors, matchers and destructors.

Consider the following datatype T:

```
data T
  = C_0 T_0_0 ... T_0_M0
  | ...
  | C_N T_N_0 ... T_N_MN
```

Its compilation to uplc results in the following: 1. one identifier per constructor: these identifiers are used as selectors; 2. a matcher function, `T_match`; 3. a destructor function, `fUnsafeFromDataT_cunsafeFromBuiltinData`, which takes uplc data and decodes it, assuming it represents `T`.

4.4.1 Selectors and matcher

The per-constructor identifiers and the matcher function are represented as follows:

```
<<< some uplc code >>>
[
  // N-nested abstraction: one variable per constructor
  [
    (lam C_0
      ...
        (lam C_N
          (lam T_match
            << some uplc code >>
          )
        )
      ...
    )
  ]
  // N-application: one abstraction per constructor
  LAM_C_0_M0
  ...
  LAM_C_N_MN
]
<<< some uplc code >>>
```

where the terms `LAM_C_I_MI`, for `0 <= I <= N`, are of the following form:

```
// MI-nested abstraction: one variable per argument of I-th constructor
(lam arg_0 ... (lam arg_MI
  // N-nested abstraction: one case variable per constructor
  (lam case_C_0
    ...
    (lam case_C_I
      // Relevant case variable applied to given arguments
      [ case_C_I arg_0 ... arg_MI ]
    )
  )
)
```

4.4.2 Destructor

The destructor function is *roughly* of the following form:

```
<<< some uplc code >>>
[
  (lam fUnsafeFromDataT_cunsafeFromBuiltinData
    <<< some uplc code >>>
```

```
  )
  // d: input data, which has to be constructor data (unConstrData)
  (lam d
    [
      // tup: a pair (cIdx:Int, cParams:list(data))
      (lam tup
        [
          // t: cParams
          (lam t
            [
              ...
               [
                  // The nested ts either isolate consecutive elements
                  // from cParams, or all equal cParams
                  (lam t
                    [
                      // index: cIdx
                      (lam index
                        CONSTRUCTOR_SWITCH
                      )
                      // cIdx
                      (delay [ (force (force (builtin fstPair))) (force tup) ])
                    ]
                  )
                  (delay [ (force (force (builtin sndPair))) (force t/tup) ])
                ]
              ...
            ]
          )
          // cParams
          (delay [ (force (force (builtin sndPair))) (force tup) ])
        ]
      )
      // (cIdx, cParams)
      (delay [ (builtin unConstrData) d ])
    ]
  )
]
```

where the `CONSTRUCTOR_SWITCH` term is an N-nested `if-then-else`, branching on the constructor index `index (cIDx)` in reverse order, and decoding and applying the parameters for each constructor appropriately:

```
[
  [
    (force (builtin ifThenElse))
    [
      (builtin equalsInteger)
      (force index)
      (con integer N)
```

```
]
// N-th constructor
(lam ds
  [
    C_N
    << decoded arg_0 >>
    ...
    << decoded arg_MN >>
  ]
)
(lam ds
  [
    [
      (force (builtin ifThenElse))
      [
        (builtin equalsInteger)
        (force index)
        (con integer (N - 1))
      ]
      // N-1-st constructor
      (lam ds [ C_N-1 ... ] )
      (lam ds
        ...
          (lam ds
            [
              [
                (force (builtin ifThenElse))
                [
                  (builtin equalsInteger)
                  (force index)
                  (con integer 0)
                ]
                // 0-th constructor
                (lam ds [ C_0 ... ] )
                // If the constructor index is not applicable,
                // then the data was not encoded properly,
                // and an error is thrown
                [ THROW_ERROR_LAM reconstructCaseError ]
              ]
            ]
            // This is effectively a force on the delay that is (lam ds ...
            unitval
          )
        ...
      )
    ]
  ]
  unitval
)
```

```
  ]
  unitval
]
```

5 Questions

Some of questions that arise at this time are the following ones.

1. Where down the pipeline do we loose uniformity, that is, we loose identifiers by DeBruijnization, optimization or inlining?
2. Is it at all possible to have a general and sound procedure to, given a contract, output its UPLC code in a uniform way?

6 Solutions?

And the following alternatives come to mind:

1. Work at PIR level? Maybe if we develop a K semantics for PIR it will allow us to have access to a more structured code. This of course would only make sense if we have a uniform use of the fixed point operator, meaningful identifiers and the Scott encoding.
2. Work at Typed Plutus Core Level? The same reasoning as above.
3. Implement our own PlutusTx compiler? This would of course give us full control of the generated UPLC code but it wouldn't be code generated by IOG's compiler which may differ in meaningful ways.