Express: Applications of Dynamically Typed Haskell Expressions

Rudy Matela
rudy@matela.com.br

Abstract

This paper presents Express, a library for manipulating dynamically typed Haskell expressions involving function application and variables. Express works as a wrapper around the `Data.Dynamic` module and provides additional features such as: explicit encoding of function applicaion thus delayed application between values, support for variable placeholders and expression matching. This paper shows these additions make this library useful in generating program specifications, automated testing and program synthesis.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**.

Keywords: functional expressions, program specification, automated testing, program synthesis, Haskell.

ACM Reference Format:
Rudy Matela. 2021. Express: Applications of Dynamically Typed Haskell Expressions. In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3471874.3472986

1 Introduction

Haskell programmers can use the `Data.Dynamic` module when they need to encode *dynamically typed values*. This module provides means to: convert any value to a value of the `Dynamic` type; perform function application between `Dynamic` values; and eventually evaluate a `Dynamic` value back into its conventional Haskell type.

In this paper, we extend `Data.Dynamic` by defining a wrapper library called Express that handles *dynamically typed expressions* with: explicit encoding of function application, delayed evaluation of applications, support for variables, pretty-printing and expression matching.

To encode a heterogeneous list of values using Express, one can write:

```
xs :: [Expr]
xs = [ val True
     , val (1 :: Int)
     , val (2 :: Int)
     , value "&&" (&&)
     , value "abs" (abs :: Int -> Int) ]
```

We use the `val` and `value` functions to convert any monomorphically typed value into a value of the `Expr` type — `val` is used on `Show` instances and `value` needs a string representation (§3). Using `evaluate` on our heterogeneous list, we attain the following results:

```
> map evaluate xs :: [Maybe Int]
[Nothing,Just 1,Just 2,Nothing,Nothing]
```

So far, Express works exactly as `Data.Dynamic`. The same behaviour can be achieved replacing functions by `Dynamic` equivalents. Express differs in how it treats applications.

Take for example the following expression that lists applications between values of the above `xs` list:

```
> catMaybes [f $$ x | f <- xs, x <- xs]
[ (True &&) :: Bool -> Bool
, abs 1 :: Int
, abs 2 :: Int ]
```

Now we see a key difference from `Data.Dynamic` (§2): resulting expressions are left unevaluated and are pretty-printed (§3). The items of the the above result list are of the `Expr` type. The above listing is from a real REPL session with the result formatted to multiple lines.

Express' representation allows for expressions containing variables (§4) and supports expression matching (§6). Express also provides a way to deeply encode values as applications of constructors (§7). This specific collection of features may seem arbitrary, but they allow Express to be useful in a few concrete applications:

- generating equalities and test properties (§5);
- generalizing counterexamples of testing (§8);
- synthesizing programs (§9).

The point is not that the Express library improves results for these applications but rather that it makes the implementation easier, shorter and more elegant (§10,§11). Using Express, prototypes for each of the above applications have less than 70 lines of code! (§13)

1.1 Contributions

The contributions of this paper are:

1. a simple representation of dynamically typed expressions involving function application and variables (§3);
2. a collection of methods and techniques for manipulating this representation, including:
   - evaluation and pretty-printing (§3);
   - variable replacement (§4);
   - expression matching (§6); and
   - deep encoding of values (§7);
3. the design of the Express library, which implements these methods in Haskell (§3, §4, §6, §7);
4. a selection of small case-studies investigating the effectiveness these methods in a few applications (§5, §8, §9) showing that this specific combination of techniques is useful in practice;
5. an analysis of how impactful was Express in simplyfying two full featured libraries (§10);
6. comparative analysis with other possible encodings of expressions and related libraries (§11).

Despite the Haskell setting of the implementation and case-studies, we expect similar techniques to be applicable in other functional programming languages with support for algebraic data types and dynamically typed values.

2 The Data.Dynamic Library

Before we examine Express in §3, §4, §6 and §7, it is useful to review the Data.Dynamic library.

The function toDyn converts any monomorphic value to a Dynamic value:

```
toDyn :: Typeable a => a -> Dynamic
```

For example:

```
dynFalse, dynZero :: Dynamic
dynFalse  =  toDyn False
dynZero   =  toDyn (0 :: Int)
```

The Show instance for Dynamic values simply shows the type. In a REPL session:

```
> dynFalse
<<Bool>>
```

We can evaluate values back into their original types with fromDynamic. It returns Just a value when the types do match and Nothing when they do not:

```
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

For example:

```
> fromDynamic dynFalse :: Maybe Bool
Just False
```

We can perform applications between Dynamic values with dynApply:

```
dynApply :: Dynamic -> Dynamic -> Maybe Dynamic
```

Using Dynamic, we can replicate the first example given in the introduction. A heterogeneous list is declared like so:

```
xs :: [Dynamic]
xs  =  [ toDyn True
       , toDyn (1 :: Int)
       , toDyn (2 :: Int)
       , toDyn (&&)
       , toDyn (abs :: Int -> Int) ]
```

It can have its values evaluated at a given type:

```
> map fromDynamic xs :: [Maybe Int]
[Nothing,Just 1,Just 2,Nothing,Nothing]
```

Results are less interesting when we list applications between values of our heterogeneous list:

```
> catMaybes [dynApply f x | f <- xs, x <- xs]
[<<Bool -> Bool>>, <<Int>>, <<Int>>]
```

As opposed to what we saw in §1, we now only see the types. In Data.Dynamic values are opaque unless evaluated at their correct types.

3 Encoding Expressions

Now we begin examining Express and its basic building blocks to encode Haskell expressions.

As stated in the introduction (§1), we encode values in the Expr type. It is declared like so:

```
data Expr  =  Value String Dynamic
           |  Expr :$ Expr
```

Atomic Values are stored as a String representation paired with a Dynamic value. We allow applications between expressions. Types of expressions are stored internally in the Dynamic value. Type-safety is provided by means of an application function ($$) defined later.

The smart-constructor value takes a string representation and an object and returns an Expr:

```
value :: Typeable a => String -> a -> Expr
value s x  =  Value s (toDyn x)
```

For example:

```
plus :: Expr
plus  =  value "+" ((+) :: Int -> Int -> Int)
```

Using value, we implement val that takes a Showable object and returns an Expr:

```
val :: (Typeable a, Show a) => a -> Expr
val x  =  value (show x) x
```

For example:

```
false, zero, one :: Expr
false  =  val False
zero   =  val (0 :: Int)
one    =  val (1 :: Int)
```

The Show instance of Exprs pretty-prints them[1]:

```
> plus :$ one
(1 +) :: Int -> Int
> plus :$ one :$ zero
1 + 0 :: Int
```

Exprs can be evaluated by the following function:

```
evaluate :: Typeable a => Expr -> Maybe a
evaluate e  =  toDynamic e >>= fromDynamic
  where
  toDynamic (Value _ x)  =  Just x
  toDynamic (e1 :$ e2)   =  do
    v1 <- toDynamic e1
    v2 <- toDynamic e2
    dynApply v1 v2
```

For example:

```
> evaluate (plus :$ one :$ zero) :: Maybe Int
Just 1
> evaluate false :: Maybe Bool
Just False
```

For convenience, Express also exports eval which takes a default value and evl which raises an error when the underlying value is not of the right type.

The :$ constructor permits ill-typed expressions:

```
> plus :$ one :$ false
1 + False :: ill-typed
```

To avoid ill-typed expressions, we should use $$:

```
($$) :: Expr -> Expr -> Maybe Expr
e1 $$ e2 | isIllTyped e  =  Nothing
         | otherwise     =  Just e
  where  e  =  e1 :$ e2
```

The isIllTyped function is implemented using functions from Data.Dynamic and Data.Typeable. Its definition is omitted here but it is straightforward. The function $$ returns Nothing when the types do not match:

```
> plus $$ false
Nothing
```

When the types do match, $$ returns Just the resulting application:

```
> plus $$ one
Just ((1 +) :: Int -> Int)
```

The function $$ is not only a convenience, but forms a core part of how we enumerate expressions later on in an example application (§5.1).

[1]Applications are pretty-printed as infix when the string-encoded function name starts with an operator character as defined in the Haskell Report [13, 20].

Though Expr values are dynamically typed at runtime, their types are always there and can be queried by the function typ :: Expr -> TypeRep:

```
> typ (plus :$ one)
Int -> Int
> typ (plus :$ one :$ zero)
Int
```

The Expr type is an instance of Eq and Ord. In both instances, Expr values are compared by their structure, string representation and type:

```
> plus :$ zero :$ one == plus :$ zero :$ one
True
> plus :$ zero :$ one == plus :$ one :$ zero
False
```

Express does not provide any direct way to encode lambdas, variable capturing or (case) pattern matching in its Expr type. Even with these limitations, it is effective in the example applications we explore later on in §5, §8, §9 and §10.

4 Encoding Variables

Because Express allows us to encode *expressions* and not only atomic values (as in Data.Dynamic), we can encode dynamically typed *variables* inside Exprs. We represent variables like so:

```
var :: Typeable a => String -> a -> Expr
var s a  =  value ('_':s) (err `asTypeOf` a)
  where  err  =  error "..."
```

Variables are atomic values whose string representation starts with an underscore and whose Dynamic value is an error value[2] that is present to carry the variable type.

Here is how to encode two variables as Exprs:

```
xx, yy :: Expr
xx  =  var "x" (undefined :: Int)
yy  =  var "y" (undefined :: Int)
```

They can be placed inside expressions and are pretty printed:

```
> xx
x :: Int
> plus :$ xx :$ one
x + 1 :: Int
```

Variables would not be useful if we could not replace them with other values or subexpressions, so Express provides the operator //- to replace them:

```
(//-) :: Expr -> [(Expr,Expr)] -> Expr
```

[2]Choices, choices... Here we could have chosen to represent variables with an additional constructor on the Expr type: Var String Typerep. The implementation presented here makes matters simpler in terms of evaluating, matching and comparing Exprs. The argument of var could have also been of the Proxy type, the version here using a simple undefined proxy was chosen for backwards compatibility.

Though this operator is able to replace any terminal atomic `Value`s, it is most useful when replacing variables. For example:

```
> plus :$ xx :$ one //- [(xx,zero)]
0 + 1 :: Int
> plus :$ yy :$ xx //- [(xx,zero), (yy,one)]
1 + 0 :: Int
```

4.1 Typed Holes

We also allow for typed holes to denote incomplete expressions. We represent them as variables without names:

```
hole :: Typeable a => a -> Expr
hole a  =  var "" (err `asTypeOf` a)
```

Here is how to declare a hole of the `Int` type:

```
i_ :: Expr
i_  =  hole (undefined :: Int)
```

Holes are pretty-printed as underscores:

```
> plus :$ i_ :$ one
_ + 1 :: Int
```

Given an expression with holes, Express has a function that lists canonical variations of variable assignments:

```
> canonicalVariations $ plus :$ i_ :$ i_
[ x + y :: Int, x + x :: Int ]
```

There are two ways to assign two variable places canonically: with two different variables or twice with the same, other variations are non-canonical. For three holes, there are five combinations:

```
  x+(y+z)  x+(y+x)  x+(y+y)  x+(x+y)  x+(x+x)
```

Canonical variations will be useful later on in a couple of example applications (§5 and §8).

We finish the description of the Express library for now. The next section (§5) describes a proof-of-concept application. We will take a look at other functionalities of Express again in §6 and §7.

5 μ-Speculate: Conjecturing Equations

In order to demonstrate how Express is useful, this section examines a simple example application called μ-Speculate (micro Speculate). This application is capable of conjecturing equations about a collection of primitive definitions based on the results of testing. μ-Speculate is a simplified reconstruction of the full-featured Speculate tool [1, 4], discussed later in §10.

By enumerating (§5.1) then equating (§5.2), we are able to list equations involving ground expressions (§5.3). By replacing variables with enumerated test values (§5.4), we are able to list equations involving variables (§5.5). Then we filter of redundant equations to arrive at our final implementation (§5.6).

5.1 Enumerating Expressions

By leveraging the application smart-constructor $$ (§3) and some enumerative functionality from LeanCheck [1] we can implement a function with the following type signature:

```
expressionsT :: [Expr] -> [[Expr]]
```

Given a set of primitive expressions, this function enumerates all possible type-correct expressions both in size-order and grouped by size. At any point in the enumeration, new expressions are built from existing type correct expressions using $$. Expressions that are not type-correct are discarded as soon as they are found. The result is a usually infinite list of finite lists. Here is an example application of `expressionsT`:

```
> expressionsT [ var "x" (undefined :: Int)
>              , val (0::Int)
>              , val (1::Int)
>              , value "+" ((+) :: Int -> ...)
>              , value "*" ((*) :: Int -> ...) ]
[ [ x :: Int
  , 0 :: Int
  , 1 :: Int
  , (+) :: Int -> Int -> Int
  , (*) :: Int -> Int -> Int
  ]
, [ (x +) :: Int -> Int
  , (0 +) :: Int -> Int
  , (1 +) :: Int -> Int
  , (x *) :: Int -> Int
  , (0 *) :: Int -> Int
  , (1 *) :: Int -> Int
  ]
, [ x + x :: Int
  , x + 0 :: Int
  , x + 1 :: Int
  , ... {- 15 Exprs omitted here -}
  ]
, [ ((x + x) +) :: Int -> Int
  , ((x + 0) +) :: Int -> Int
  , ... {- 34 Exprs omitted here -}
  ]
, ... {- infinite list -}
]
```

Expressions above are shown as they would when pretty-printed by the show function.

Using `expressionsT`, we can implement a function that enumerates expressions up to an arbitrary size limit of 5 by:

```
candidateExprsFrom :: [Expr] -> [Expr]
candidateExprsFrom  =  concat . take 5
                    .  expressionsT
```

5.2 Equating Expressions

Consider the following function `-==-` which given two `Expr`s returns the encoded application of `==` between them for a hardcoded selection of types:

```
(-==-) :: Expr -> Expr -> Expr
ex -==- ey  =  head $
  [eqn | eq <- eqs
       , let eqn = eq :$ ex :$ ey
       , isWellTyped eqn] ++ [val False]
  where
  eqs = [ value "==" ((==)::Int->Int->Bool)
        , value "==" ((==)::Bool->Bool->Bool)
        , value "==" ((==)::[Int]->[Int]->Bool)
        , value "==" ((==)::[Bool]->...) ]
```

We default to a `False` value encoded as an `Expr` in case we do not find the appropriately typed `==`.

Here is an example application of `-==-`:

```
> (plus :$ xx :$ zero) -==- one
x + 0 == 1 :: Bool
```

Using `-==-` and `candidateExprsFrom` we can enumerate candidate equations between expressions involving a list of primitives:

```
candidateEquationsFrom :: [Expr] -> [Expr]
candidateEquationsFrom es'  =
  [e1 -==- e2 | e1 <- es, e2 <- es, e1 >= e2]
  where  es = candidateExprsFrom es'
```

The use of `>=` above avoids some redundant equations — equality is commutative.

5.3 Listing Equations between Ground Expressions

Now we can implement the first version of a function that lists equations about given primitives:

```
speculateAbout :: [Expr] -> [Expr]
speculateAbout  =  filter (eval False)
                .  candidateEquationsFrom
```

This initial version only works when primitives do not include variables, like so:

```
> speculateAbout [ val ([] :: [Int])
>                , value ":" ((:) :: Int->...)
>                , value "++" ((++)::[Int]...) ]
[  [] == [] :: Bool
,  [] ++ [] == [] :: Bool
,  [] ++ [] == [] ++ [] :: Bool
,  ... {- 7 equations omitted -}
]
```

Not very interesting or useful output so far.

5.4 Listing Ground Expressions

In order to get more interesting and useful output, we need to allow for variables in our equations, We define a function to list ground expressions, i.e., expressions whose variables have been replaced by ground values:

```
grounds :: Expr -> [Expr]
```

The resulting list may be infinite. The definition of `grounds` is omitted here but uses `//-` (§4) and enumerated values from LeanCheck[3]. Here are a couple of example applications:

```
> grounds $ value "not" not :$
>           var "p" (undefined :: Bool)
[not False :: Bool, not True :: Bool]
> grounds $ plus :$ xx :$ yy
[ 0 + 0 :: Int, 0 + 1 :: Int, 1 + 0 :: Int
, 0 + (-1) :: Int, 1 + 1 :: Int, ... ]
```

Using `grounds` we can implement `isTrue`, that checks if a boolean expression is true for an arbitrary value of 60 assignments of variables:

```
isTrue :: Expr -> Bool
isTrue  =  all (eval False) . take 60 . grounds
```

5.5 Equations Involving Variables

We now refine our `speculateAbout` function:

```
speculateAbout  =  discardLater isInstanceOf
                .  filter isTrue
                .  candidateEquationsFrom
```

We allow for variables by using `isTrue` and we discard later[4] equations that are instances of earlier equations using `isInstanceOf` provided by Express (§6).

We can now include a variable `xs` in the argument list of the `speculateAbout` function:

```
> speculateAbout [ var "xs" (undefined :: [Int])
>                , val ([] :: [Int])
>                , value ":" ((:) :: Int -> ...)
>                , value "++" ((++) :: ...) ]
[ xs == xs :: Bool
, xs ++ [] == xs :: Bool
, [] ++ xs == xs :: Bool
, [] ++ xs == xs ++ [] :: Bool
, xs ++ (xs ++ []) == xs ++ xs :: Bool
, ... {- 113 equations omitted -} ]
```

We start to see a few interesting equations.

[3]We use LeanCheck [1] but other property-based testing tools would work as well: QuickCheck [6–8], SmallCheck [23, 24], Feat [10, 11], or others. The function `grounds` works with the same selection of types as `-==-`.

[4]The function `discardLater` is similar to `nubBy` but it does not require the given predicate to be an equivalence.

5.6 Non-Redundant Equations

Consider the following idea [26]: we can begin by testing the one-variable instance of an equation first and only if it is true we test the multiple-variable instance of an equation. This works because for an expression of boolean result with multiple variables (of each type) to be true, its instance where there is only one variable must be true as well. In symbolic terms:

$$\forall x, y, z \ldots\ p(x, y, z, \ldots) \Rightarrow \forall x.\ p(x, x, x, \ldots)$$

$$\neg\forall x\ p(x, x, x, \ldots) \Rightarrow \neg\forall x, y, z \ldots\ p(x, y, z, \ldots)$$

Based on the above observation, we now implement the final version of our speculateAbout function:

```
speculateAbout :: [Expr] -> [Expr]
speculateAbout  =
     discardLater canBeSimplifiedBy
  .  discardLater isInstanceOf
  .  concatMap trueCanonicalVariations
  .  discardLater
       (\e1 e2 -> isntIdentity e2
               && e2 `isInstanceOf` e1)
  .  sort
  .  filter isTrue
  .  candidateEquationsFrom
  where
  e1 `canBeSimplifiedBy` e2  =
    isRule e2 && e1 `hasInstanceOf` lhs e2
  trueCanonicalVariations  =
       discardLater isInstanceOf
    .  filter isTrue
    .  filter isntIdentity
    .  canonicalVariations
```

Now, to conjecture equations about [], : and ++, we do:

```
speculateAbout
  [ hole (undefined :: Int)
  , hole (undefined :: [Int])
  , val ([] :: [Int])
  , value ":" ((:) :: Int -> ...)
  , value "++" ((++) :: [Int] -> ...) ]
```

Above we provide two holes indicating that we want variables of their type to appear in equations. The above call returns the following equations:

```
xs ++ []  ==  xs
[] ++ xs  ==  xs
[x] ++ xs  ==  x:xs
(x:xs) ++ ys  ==  x:(xs ++ ys)
(xs ++ ys) ++ zs  ==  xs ++ (ys ++ zs)
```

in less than 1s. Notably the 4th equation is a possible recursive step in an implementation of ++.

Here is what μ-Speculate says about 0, + and abs:

```
abs 0  ==  0
x + 0  ==  x
0 + x  ==  x
abs (abs x)  ==  abs x
x + y  ==  y + x
abs (x + y)  ==  abs (y + x)
abs (x + abs x)  ==  x + abs x
abs (abs x + x)  ==  x + abs x
abs x + abs x  ==  abs (x + x)
(x + y) + z  ==  x + (y + z)
```

along with 14 redundant equations omitted here.

Equations produced by μ-Speculate are just conjectures. They were not proven to be true, just *tested* to be true.

There are limitations to μ-Speculate:

- there are still quite a few redundant equations;
- number of tests and maximum equation size not configurable;
- the supported types are hardcoded in -==- and grounds;
- performance.

μ-Speculate is a simplified reconstruction of the full-featured Speculate tool [1, 4] that is not constrained by the above limitations. The generation of equations based on the results of testing was originally described in an ealier paper about the QuickSpec tool [9, 26].

The point of μ-Speculate is not to be the best equational generator but to show that Express works well as the framework on which to build equational generators. The whole implementation is only 67 lines of code. The full code of μ-Speculate including omitted functions is available in the Express package (§13).

6 Matching Expressions

In the previous section (§5) we used isInstanceOf to discard redundant equations. This section discusses this functionality in more detail.

Express provides a function match that returns a *structural* match between two expressions. When the match is possible, this function returns a list of assignments of variables of the second expression to subexpressions of the first:

```
match :: Expr -> Expr -> Maybe [(Expr,Expr)]
```

For example, the expression encoding $0 + 1$ matches the expression $x + y$ by assigning x to 0 and y to 1:

```
> match (plus :$ zero :$ one) (plus :$ xx :$ yy)
Just [(y :: Int,1 :: Int), (x :: Int,0 :: Int)]
```

On the other hand, the expression $0 + 1$ does not match the expression $x + x$ as it would require assigning the variable x two different values:

```
> match (plus :$ zero :$ one) (plus :$ xx :$ xx)
Nothing
```

Matches are structural: applications have to happen in the same place; types and string representations of corresponding terminal values must be equal; and types of variables must match corresponding sub-expressions. Consequently, an expression encoding $(0 + 1) + 2$ matches both $(x + y) + z$ and $x + y$ but not $x + (y + z)$.

The implementation of `match` is omitted here but it is straightforward. It maintains a list of assignments from variables to values and this list is updated as both expressions are traversed.

Using `match`, the implementation of the `isInstanceOf` function is trivial:

```
isInstanceOf :: Expr -> Expr -> Bool
e1 `isInstanceOf` e2  =  isJust $ match e1 e2
```

When there is a match this function returns `True`, otherwise it returns `False`.

The functions `match` and `isInstanceOf` are not only useful to discard redundant equations (§5). but can be applied to discard redundant expressions at the moment of enumeration or even to perform rewrites (§10).

7 Deeply Encoding Values as Applications of Constructors

Now, we examine another feature of Express, deep encoding of values. This will be useful in the next example application (§8).

The function `val` encodes values atomically, e.g., the following two are equivalent:

```
val [1,2,3 :: Int]
value "[1,2,3]" [1,2,3 :: Int]
```

To deeply encode values we define a typeclass of values that can be expressed as applications of constructors:

```
class Typeable a => Express a where
  expr :: a -> Expr
```

Atomic values are still encoded atomically:

```
instance Express Bool where  expr = val
instance Express Int  where  expr = val
instance Express Char where  expr = val
```

A deeply encoded pair is the pair constructor represented as an `Expr` applied to the deep encoding of the two elements:

```
instance (Express a, Express b)
      => Express (a,b) where
  expr (x,y)  =  value "," ((,) ->>: (x,y))
              :$ expr x
              :$ expr y
```

Above, the operator `->>:` is a type restricted version of the `const` function that binds the result type of the pair constructor `(,)` to match `x` and `y`.

A deeply encoded list is either the `Expr` representation of the atomic nil `[]` value or the list constructor `:` represented as an `Expr` applied to the deep encoding of the head and tail.

```
instance Express a => Express [a] where
  expr xs  =  case xs of
              [] -> val xs
              (y:ys) -> value ":" ((:) ->>: xs)
                     :$ expr y
                     :$ expr ys
```

A similar pattern goes for other types. Express provides instances for most standard Haskell types and a facility to automatically derive instances for user-defined data types using Template Haskell [25]. Writing:

```
deriveExpress ''UserDefinedDataType
```

is enough to derive an `Express` instance for most user defined algebraic data types.

8 μ-Extrapolate: Generalizing Counterexamples

To further demonstrate the usefulness of Express, this section describes a short example application called μ-Extrapolate (micro Extrapolate). This application does property-based testing and aside from reporting a fully-defined counterexample, it reports a generalization. μ-Extrapolate is a simplified reconstruction of the full-featured Extrapolate tool [1, 3].

Given a maximum number of tests and a property, the following `counterExamples` function returns an empty list when when tests pass or a list of counterexamples deeply encoded as `Expr`s when tests fail.

```
counterExamples :: (Listable a, Express a)
                => Int -> (a -> Bool) -> [Expr]
counterExamples max prop  =
  [expr x | x <- take max list, not (prop x)]
```

We can also build a version that returns `Just` the first counterexample if any is found or `Nothing` otherwise:

```
counterExample m  =
  listToMaybe . counterExamples m
```

Here are some simple example applications:

```
> counterExample 100 $ \(x,y) -> x + y == y + x
Nothing
> counterExample 100 $ \x -> x == x + x
Just (1 :: Integer)
> counterExample 100 $
>     \xs -> nub xs == (xs :: [Int])
Just ([0,0] :: [Int])
```

Addition is commutative; and doubling a number or nubbing a list are not identities.

To compute candidate generalizations from a given counterexample deeply encoded as an Expr, we use the following:

```
candidateGeneralizations :: Expr -> [Expr]
candidateGeneralizations  =
  concatMap canonicalVariations . g
  where
  g e@(e1 :$ e2)  =
    [holeAsTypeOf e | isListable e] ++
    [g1 :$ g2 | g1 <- g e1, g2 <- g e2] ++
    map (:$ e2) (g e1) ++
    map (e1 :$) (g e2)
  g e | isVar e    =  []
      | otherwise  =  [ holeAsTypeOf e
                      | isListable e ]
```

Above, the isListable function returns whether we can enumerate variables of the given variable type with grounds.

Our candidate generalizations are listed in non-increasing order of generality. Candidate generalizations for [0] are:

```
> candidateGeneralizations $ expr [0::Int]
[ xs :: [Int]
, x:xs :: [Int]
, [x] :: [Int]
, 0:xs :: [Int] ]
```

The candidate generalizations for [0,0] are:

```
    xs       [x,y]    0:xs
    x:xs     [x,x]    0:x:xs
    x:y:xs   x:0:xs   [0,x]
    x:x:xs   [x,0]    0:0:xs
```

For a given maximum number of tests, property and counterexample, the following function returns a counterexample along with generalizations if any is found. Generalizations are selected when they *fail* for *all* tests!

```
counterExampleAndGeneralizations
  :: (Listable a, Express a)
  => Int -> (a -> Bool) -> [Expr]
counterExampleAndGeneralizations maxTests prop =
  case counterExamples maxTests prop of
  [] -> []
  (ce:_) -> ce : gens ce
  where
  gens ce  =  discardLater isInstanceOf
    [ g | g <- candidateGeneralizations ce
        , all (not . prop . evl)
              (take maxTests $ grounds g) ]
```

The previously defined function grounds is recycled from our previous example application (§5). Above, evl (§3) converts the enumrated ground Expr back into the argument-type of the property.

By pattern matching on the result of the above function it is straightforward to define a check function. It tests a property for 500 arguments and reports a counterexample and its generalizations when any is found:

```
check :: (Listable a, Express a)
      => (a -> Bool) -> IO ()
```

Now we can find counterexamples and their generalizations. See:

```
> check $ \xs -> sort (sort xs)
>             == sort (xs :: [Int])
+++ Tests passed.
```

The above property is correct, sort is idempotent.

```
> check $ \xs -> length (nub xs)
>             == length (xs :: [Int])
*** Falsified, counterexample:  [0,0]
               generalization:  x:x:xs
```

The above property is incorrect for [0,0] and any counterexample of the form x:x:xs: [1,1], [2,2], [0,0,1], etc. In general, the property is incorrect whenever there are repeated elements in the given list. The generalized counterexample hints at this.

Here are a couple more simple examples:

```
> check $ \x -> x == x + (1 :: Int)
*** Falsified, counterexample:  0
               generalization:  x
```

The above property fails for any argument value.

```
> check $ \(x,y) -> x /= (y :: Int)
*** Falsified, counterexample:  (0,0)
               generalization:  (x,x)
```

The above property fails when the argument numbers are repeated.

The produced counterexamples are known to falsify the property, however their generalizations are conjectures based on the results of testing.

μ-Extrapolate is a simplified reconstruction of the full-featured Extrapolate tool [1, 3] which also supports conditional generalizations. The original paper about Extrapolate discusses how generalizations can help inform programmers about the actual source of a bug. The idea of generalizing counterexamples of property testing in Haskell was presented in an earlier paper [22] about the SmartCheck tool.

The point of μ-Extrapolate is not to be the best counterexample generalizer but to show that Express works well as the framework on which to build libraries able to generalize counterexamples. The whole implementation is only 51 lines of code. The full code of μ-Extrapolate including omitted functions is available in the Express package (§13).

9 μ-Conjure: Program Synthesis

In this section we use Express to implement our last example application called μ-Conjure (micro Conjure). It is capable of synthesizing (or conjuring) Haskell functions out of partial definitions.

We start with a partial definition:

```
square :: Int -> Int
square 1  =  1
square 2  =  4
square 3  =  9
square 4  =  16
```

and a list of primitives that are allowed in the synthesized definition:

```
primitives :: [Expr]
primitives  =  [ val (0 :: Int)
               , val (1 :: Int)
               , value "+" ((+) :: Int -> ...)
               , value "*" ((*) :: Int -> ...) ]
```

Then we declare a function that, given a name and a function, builds an application with variables encoded as an `Expr`:

```
application :: Typeable f
            => String -> f -> [Expr] -> Expr
```

For example:

```
> application "square" square primitives
square x :: Int
> application "&&" (&&) primitives
p && q :: Bool
```

These applications will serve as the left hand side of our synthesized definition. Here we omit the code of `application` but it is straightforward to build using combinators from the Express library.

Now we can define a function that lists matching implementations. Given a name, a function and a list of primitive values, `conjureImpls` returns a list of matching implementations using the given primitive values. If no matching implementation is found, `conjureImpls` returns an empty list.

```
conjureImpls :: Typeable f
             => String -> f -> [Expr] -> [Expr]
conjureImpls nm f ps  =
  [ appn -==- e
  | e <- candidateExprsFrom (exs ++ ps)
  , isTrue (appn -==- e) ]
  where
  appn  =  application nm f ps
  (ef:exs)  =  unfoldApp appn
  isTrue e  =  all (errorToFalse . eval False)
            .  map (e //-)
            $  definedBinds appn
```

The `conjureImpls` function first builds a full application of the function (`appn`). Then, for all candidate expressions it can generate including primitives and variables in the application, it returns equations between the full application and the candidate (`appn -==- e`) whenever they are is tested to be true (`isTrue`). The truth check here is only for the defined assignments (`definedBinds`) of the argument function.

The function `unfoldApp` is available from the Express library and unfolds an application of `Expr`s into a list. The `definedBinds` function was created for μ-Conjure and enumerates assignments for which the given application is defined.

For convenience, it is nice to define a function `conjure` that instead of returning a list of `Expr`s, simply prints the first matching implementation. We can finally synthesize our `square` function:

```
> conjure "square" square primitives
square :: Int -> Int
square x  =  x * x
```

Here is a partial definition of a factorial function:

```
factorial :: Int -> Int
factorial 1  =  1
factorial 2  =  2
factorial 3  =  6
factorial 4  =  24
```

By including `foldr` and `enumFromTo` in the primitives, μ-Conjure is able to produce the following:

```
factorial :: Int -> Int
factorial x  =  foldr (*) 1 (enumFromTo 1 x)
```

μ-Conjure also works for list-processing functions. given the following partial definition of append (++):

```
(+++) :: [Int] -> [Int] -> [Int]
[x] +++ [y]  =  [x,y]
[x,y] +++ [z,w]  =  [x,y,z,w]
```

A call to `conjure` including the list constructor and `foldr` in the primitives argument is able to produce a correct append definition:

```
(+++) :: [Int] -> [Int] -> [Int]
xs +++ ys  =  foldr (:) ys xs
```

μ-Conjure has some limitations:

- no recursion is allowed;
- supported types are once again hardcoded;
- brute-force search without pruning;
- runtime is reasonable only for less than a dozen primitives and up to functions with 7 applications. This means several relatively simple functions are out of reach.

A full featured version that addresses these limitations is left as future work (§12).

Table 1. Number of lines of code in Speculate and Extrapolate before and after the switch to using Express. Figures exclude blank lines and comments.

tool/lib	#-lines	
Speculate v0.3.5	3181	before Express
Speculate v0.4.0	2570	after Express, −20%
Speculate v0.4.10	2560	latest version
Extrapolate v0.3.3	1359	before Express
Extrapolate v0.4.0	958	after Express, −30%
Extrapolate v0.4.6	947	latest version

μ-Conjure has much poorer performance than MagicHaskeller [14–17] or Igor II [12], two other tools that are able to generate Haskell functions. However, the point of μ-Conjure is not to be an efficient program generator but to show that Express can work as a framework on which to build libraries able that synthesize programs. The whole implementation of Express is only 64 lines of code, whereas MagicHaskeller is over 9000! Whether our approach here could scale up to be on par with MH is an open question (§12). The full code of μ-Conjure including omitted functions is available in the Express package (§13).

10 Beyond Simple Proofs of Concept

In §5, §8 and §9 we see Express applied in three simple proofs of concept. How does it fare when applied in the implementation of full featured libraries and programs?

Express was born as an internal module of the Speculate tool [4]. This module was used later on the Extrapolate tool [3]. From this usefulness in two different applications came the idea of providing Express as a standalone library.

The original Speculate and Extrapolate libraries have since been changed to explicitly use the Express library as a dependency. Though this is subjective, the use of Express did make the code more elegant and organized. The earlier libraries are now easier to maintain, as one does not need to worry about the lower level machinery. As a more objective measure, the number of lines of code in the implementation of Speculate and Extrapolate have been significantly reduced by 20% and 30% respectively (Table 1).

In their respective original papers, Speculate and Extrapolate were demonstrated to work in real applications with reasonable performance. After the change to use Express, Speculate and Extrapolate had no decrease in performance. Most of their built-in benchmarks had no increase in runtime from the version without Express compared to the version that uses Express. This is shown in Table 2. Since there were unrelated changes in the same development period, we cannot claim that Express is the central reason for runtime change in any benchmark, only that the tools still work with reasonable runtime after the change.

Table 2. Runtimes for accompanying benchmarks of Speculate and Extrapolate comparing old versions that did not use Express with new versions that do use Express.

tool / benchmark	old	new
Speculate	v0.3.5	v0.4.0
insertsort	6.3s	7.5s
list	4.0s	2.5s
plus-abs	3.9s	3.5s
binarytree	2.2s	1.7s
digraphs	1.4s	1.5s
regexes	8.5s	7.6s
Extrapolate	v0.3.3	v0.4.0
sorting	23.6s	23.8s
calculator	0.4s	0.4s
parser	6.6s	6.3s
heap	2.9s	2.7s
int	2.6s	2.8s
list	7.3s	6.8s
word	8.3s	7.9s

Benchmarks in Table 2 were compiled with GHC version 8.10.4 and `-O2` flag under Linux on a PC with a 2.2Ghz 4-core processor and 16GB of RAM. The number of cores here is of less relevance as both Speculate and Extrapolate run on a single thread in a single process. The given figures are averages of 60 runs rounded to one decimal place. Default Extrapolate settings changed between versions and were normalized to the ones of v0.4.0. Most benchmarks are described in the original papers [3, 4]. Benchmark sources are available in each tool's respective packages.

The μ-Speculate (§5) and μ-Extrapolate (§8) examples are simplifications and reconstructions of the original full-featured libraries. How the lower-level machinery worked was not detailed in the original papers and was left out as implementation detail – there algorithms and methods are described in a higher level, here we examine the lower-level machinery.

In Speculate there is a more efficient implementation of expression enumeration (§5.1): there we prune redundant expressions during enumeration using discovered equations themselves by treating them as term-rewriting rules. This is done with the help of the `match` and `isInstanceOf` functions described in §6. The higher level algorithm that does this is described in the original paper about Speculate [4]. This algorithm is in turn based on the original algorithm of QuickSpec [9, 26]. In both Speculate and Extrapolate, there are ways to collect and maintain lists of typeclass instances for Expr values, solving the issue of hardcoded types in `grounds` and `-==-`.

Table 3. Number of constructors in different representations of Haskell expressions from a few packages.

package	version	module	datatype	#-cons	
express	v0.1.10	Data.Express	Expr	2	
extrapolate	v0.4.4	Data.Express	Expr	2	(uses Express)
speculate	v0.4.6	Data.Express	Expr	2	(uses Express)
speculate (old)	v0.3.5	Test.Speculate.Expr.Core	Expr	3	
quickspec	v2.1.5	QuickSpec.Internal.Term	Term	3	
template-haskell	v2.17.0.0	Language.Haskell.TH	Exp	29	
haskell-src	v1.0.3.1	Language.Haskell.Syntax	HsExp	27	
haskell-src-exts	v1.23.1	Language.Haskell.Exts.Syntax	Exp	56	
MagicHaskeller	v0.9.6.7	MagicHaskeller.CoreLang	CoreExpr	21	
Igor II	v0.8.0	Syntax.Expressions	TExp	8	

Table 4. Presence of features in three libraries that encode Haskell expressions: ●=yes; ○=no.

	Express	Template Haskell	haskell-src(-exts)
"`eval`-style" evaluation	●	○	○
expr-matching	●	○	○
compile time splicing	○	●	○
direct encoding of:			
– function application	●	●	●
– variables	●	●	●
– lambdas	○	●	●
– case pattern-matching	○	●	●

11 Other Representations of Haskell Expressions

In Express we took a minimalist approach to encode Haskell expressions. We only used two constructors!

There are other packages that provide types that encode Haskell expressions. Some packages take a radically different approach of using dozens of constructors as they are intended for different applications: Template Haskell [25], haskell-src [21], haskell-src-exts [5] and MagicHaskeller [14–17] have expression types with 29, 27, 56 and 21 constructors respectively including explicit lambda-, case-, if- and let- expressions (Table 3). These representations are more expressive but they can be harder to use. A complete pattern match on `Exp` from the haskell-src-exts package needs at least 56 lines of code! Equivalent implementations of Express' `eval`, `evaluate`, `match` and `isInstanceOf` would be more lengthy just by the sheer number of cases to be handled. However, comparing the number of constructors is not entirely fair as these packages have inherently different goals thus different sets of features (Table 4).

Express' representation comes from the former internal representation of Speculate and has a similar number of constructors. QuickSpec [9, 26] is another tool that represents its internal expressions with a few constructors, only 3 (Table 3). QuickSpec's representation was constructed with the purpose of conjecturing equations, Express' is intended to be more general purpose.

The example applications show that one can produce interesting results with a small number of constructors (§5, §8, §9). Specifically for generation of equations and generalization of counterexamples, existing work on QuickSpec, Speculate and Extrapolate use a small number of constructors to represent expressions [1, 3, 4, 9, 26]. On the other hand, for program synthesis, representing with several constructors seems to be the norm [12, 14–17]. What the exact sweet-spot is in each application is an open ended question.

A direct and fair comparison between Template Haskell and Express is difficult as they have inherently different goals and features (Table 4). TH's `Exp` type is intended to be spliced in programs during compile time whereas Express' `Exp` type cannot be spliced directly. One can easily automate the generation of boilerplate code with TH but not with Express. Express itself uses TH to automate the generation of `Express` instances (§7). Conversely, Express provides readily available functions `evaluate` and `eval` that take an `Expr` and returns a regular Haskell value whereas TH does not.[5] The same can be said for variable substitution with `//-` or expression matching with `match` and `isInstanceOf`: there are no readily available equivalents in TH and to construct equivalents would require handling a significant portion of the 29 constructors in its `Exp` type. Again, different goals, different features.

[5]TH `Exp` values are of course eventually evaluated at a later time. Splicing happens at compile time but evaluation happens at runtime. We make no claim that evaluating TH's `Exp`s as soon as they are constructed is impossible, but one would have to take into account the GHC stage restriction: `Exp`s that are spliced must be imported and cannot defined locally.

12 Conclusions

This paper presented the Express library (§3, §4) that facilitates manipulating Haskell expressions involving function application and variables. To show that Express is useful, it was applied in three short example applications: μ-Speculate (§5); μ-Extrapolate (§8); and μ-Conjure (§9). The point of Express is to ease implementation of systems that manipulate Haskell expressions and the example applications presented here are meant to show this. We make no claim that these particular implementations are the best tools in their domains. Disregarding blank lines and comments, these applications were implemented in 67, 51 and 66 lines of code respectively.

This paper discussed the use of Express in actual full-featured libraries (§10) Speculate and Extrapolate. Newer versions were changed to use Express instead of original internal representations. Subjectively, they are now easier to maintain. Objectively, they now have fewer lines of code. Performance was not affected negatively.

In this paper we only examine a few functions in the Express library. The complete library has over a hundred functions for building, evaluating, comparing, folding, canonicalizing and matching `Exprs`. It is well documented, with over 95% Haddock documentation coverage [19] in most of the cases including small examples. Express totals around 2500 lines of code.

Future Work. We are working on an improved version of μ-Conjure (§9) called Conjure to hopefully be discussed in a future paper. Likewise as in previous work, the description of Express will be left out as implementation detail though it is the machinery that enables Conjure to work.

We hope Express can be applied in other areas: perhaps to mutation testing in Haskell [2, 10, 18]; figuring specifications about concurrent functions [27, 28]; or even another application not considered here.

13 Availability

Express is feely available with a BSD3-style license from:

- https://hackage.haskell.org/package/express
- https://github.com/rudymatela/express

The above links include the full sources of the μ-$*$ example applications. This paper describes Express version 1.0.0.

Acknowledgements

Thanks to Johannes Waldmann, José Manuel Calderón Trilla, Michael Walker, Wilhelm Bartel, and anonymous reviewers for their comments on earlier drafts.

References

[1] Rudy Matela Braquehais. 2017. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing.* Ph.D. Dissertation. University of York.

[2] Rudy Matela Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Haskell'16.* ACM, 1–12.

[3] Rudy Matela Braquehais and Colin Runciman. 2017. Extrapolate: generalizing counterexamples of functional test properties. In *IFL 2017.* ACM, 11 pages.

[4] Rudy Matela Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell'17.* ACM, 40–51.

[5] Niklas Broberg. 2008–2020. haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/haskell-src-exts.

[6] Koen Claessen. 2012. Shrinking and Showing Functions. In *Haskell'12.* ACM, 73–80.

[7] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00.* ACM.

[8] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. In *Haskell '02.* ACM, 65–77.

[9] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010.* Springer, 6–21.

[10] Jonas Duregård. 2016. *Automating black-box property based testing.* Ph.D. Dissertation. Chalmers University of Technology.

[11] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell'12.* ACM, 61–72.

[12] Martin Hofmann. 2010. IgorII - an analytical inductive functional programming system (tool demo). In *PEPM.* ACM, 29–32.

[13] Simon Peyton Jones et al. 2002. Haskell 98 Language and Libraries: The Revised Report. https://www.haskell.org/onlinereport/.

[14] Susumu Katayama. 2004. Power of Brute-Force Search in Strongly-Typed Inductive Functional Programming Automation. In *PRICAI 2004: Trends in Artificial Intelligence.* Springer, 75–84.

[15] Susumu Katayama. 2007. Systematic search for lambda expressions. In *Trends in Functional Programming (TFP2005).* Vol. 6. Intellect, 111–126.

[16] Susumu Katayama. 2008. Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening. In *PRICAI 2008: Trends in Artificial Intelligence.* Springer, 199–210.

[17] Susumu Katayama. 2010. Recent Improvements of MagicHaskeller. In *AAIP.* Springer, 174–193.

[18] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *ISSTA 2014.* ACM, 429–432.

[19] Simon Marlow. 2002. Haddock, a Haskell documentation tool. In *Haskell'02.* ACM, 78–89.

[20] Simon Marlow et al. 2010. Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010.

[21] Simon Marlow, Sven Panne, and Noel Winstanley. 2006–2019. haskell-src: Support for manipulating Haskell source code. https://hackage.haskell.org/package/haskell-src.

[22] Lee Pike. 2014. SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization. In *Haskell'14.* ACM, 59–70.

[23] Jason S. Reich, Matthew Naylor, and Colin Runciman. 2013. Advances in Lazy SmallCheck. In *IFL'13.* Springer, 53–70.

[24] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell'08.* ACM, 37–48.

[25] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. In *Haskell'02.* ACM, 1–16.

[26] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017).

[27] Michael Walker and Colin Runciman. 2018. Cheap Remarks About Concurrent Programs. In *FLOPS 2018: Functional and Logic Programming (LNCS 10818).* Springer, 264–279.

[28] Michael Stewart Walker. 2018. *Revealing Behaviours of Concurrent Functional Programs by Systematic Testing.* Ph.D. Dissertation. University of York.