Non-Deterministic Symbolic Analysis using Free Monads for Test Data Generation

Adrian Mensing

Abstract

Constraint logic programming (CLP) is a combination of two programming paradigms: constraint programming and logic programming. This combination allows us to write programs in a concise way and still be executed efficiently. CLP is commonly used for many domains, such as test data generation. A commonly used technique for doing this is by performing symbolic analysis on a program, and determine which values will cause which branches of the program to be executed. The problem however is that these symbolic execution algorithms normally analyze the program in a depth-first manner and do not treat non-determinism fairly. We will be investigating how we could improve this using the free monad and how we can use this in order to perform a breadth-first search, rather than depth-first. We will give a basic sample interpreter, which we will convert into a symbolically executable interpreter. The language will be described and represented as an interpreter written in Scala.

1 Introduction

Constraint logic programming (CLP) is a concept where logic programming is extended with concepts from the constraint programming paradigm. This allows the possibility to solve logic equations using constraint satisfaction strategies (Rossi et al., 2006). This makes CLP attractive, because it allows the user to write concise programs in a declarative way and still run efficiently. Because of this, CLP is used in many domains, such as program verification or solving scheduling problems. One specific domain that we are interested in, is test data generation.

By performing symbolic analysis on a program, we are able to determine which symbolic values will cause which part of the program to execute. This particular technique is commonly known as concolic testing (Sen, 2007). Some examples of concolic testing tools include DART (Godefroid et al., 2005), KLEE (Cadar et al., 2008), and QuickCheck (Claessen and Hughes, 2011). These tools are able to obtain good code coverage when run on normal programs, by using CLP solving methods or SMT solvers. However, these tools face a problem when encountering non-determinism or infinite recursion. Concolic testing algorithms generally perform a depth-first search over the tree of possible execution paths (Sen, 2007), and cannot handle infinite recursion and non-termination fairly. In this paper, our objective is find out how symbolic analysis of programs can be performed in a breadth-first manner, by representing these programs using free monads and how we can use CLP solving techniques in order to generate the actual test data (e.g. for unit tests). We will be exploring the free monad due to its ability to represent programs in a tree-like structure, without actually performing the computations.

2 The Basic Interpreter

In this chapter, we will introduce a simple interpreter for a language with support for pattern matching. We will discuss the expressions and values that are necessary for achieving this, as well as the evaluation strategy, followed by an implementation of such an interpreter.

2.1 Definition of Expressions

For our basic interpreter, a program is defined as a single expression. One of the interesting features that most functional languages include, is pattern matching. In order to achieve that, we define that an expression that can be represented using the following case classes:

```
trait Expr

object Expr {
  case class Var(name: String)                                        extends Expr
  case class Val(value: Any, args: List[Expr])                        extends Expr
  case class Function(arg: Var, body: Expr)                           extends Expr
  case class Letrec(bindings: List[(Var, Function)], body: Expr)      extends Expr
  case class Apply(body: Expr, arg: Expr)                             extends Expr
  case class Case(pattern: Pattern, body: Expr)                       extends Expr
  case class Match(arg: Expr, cases: List[Case])                      extends Expr
}
```

A `Var` expression simply represents a variable with the given name. Values are represented with `Val` expressions, which basically acts as a value constructor. Integers in our language are represented as `Val("Int", List(Val(1, Nil)))`. Using this method of representation, we could also represent the list `[1, 2, 3, 4]` using `Cons` and `Empty` in the following way:

```
    Val("Cons", List(Val("Int", List(Val(1, Nil))),
        Val("Cons", List(Val("Int", List(Val(2, Nil))),
            Val("Cons", List(Val("Int", List(Val(3, Nil))),
                Val("Cons", List(Val("Int", List(Val(4, Nil))),
                Val("Empty", Nil)))
            ))
        ))
    ))
```

Which roughly translates to `Cons(1, Cons(2, Cons(3, Cons(4, Empty))))`. For our language, we would also need to find a way to define functions. This is done using the `Function` expression, which takes only one argument (`arg`) and a `body`, which is the body of the function, where the argument is bound. These functions can then be used to define `Letrec` expressions. A `Letrec` expression takes in a list of a `(Var, Function)` tuples, where for each tuple, the function is bound to the corresponding variable. Using these functions can be done by using `Apply` expressions, which applies the given argument on the given body.

For the pattern matching feature, we have two different case classes. As we can see, a `Case` expression accepts a `Pattern` class. This class is used to define patterns, such that the interpreter can match values and variables with the given value. A `Pattern` class has only two cases. Namely when the value is known (`ValP`), and when the value is not known (`VarP`). These case classes are defined in the following way, similar to a normal expression:

```
sealed trait Pattern

object Pattern {
  case class ValP(value: Any, args: List[Pattern])  extends Pattern
  case class VarP(name: String)                     extends Pattern
}
```

Both case classes act similar to the equivalent `Expr` variant. When such a pattern can be matched by the given value, the `body` expression is then executed. In the `body` expression, new variables can be used that were introduced during the pattern matching (bound using `VarP` expressions).

These `Case` expressions are then used to form `Match` expressions, which matches the given argument and executes the body of the first `Case` expression where the argument is able match the pattern of that `Case` expressions.

2.2 The Basic Interpreter

Interpretation of expressions is done using a well-known tradition, which is by interpreting the expressions with an *environment*, which maps variables to their values (Landin, 1964). The interpretation method (`interp`) interprets an expression using lazy evaluation. In order to do this, we use *closures*, which stores a function together with an environment (Sussman and Steele, 1998). These closures are represented using `ClosV` instances. The interpretation method simply returns a `Value` instance, which is defined using the following case classes:

```
sealed trait Value

object Value {
  case class ValV(x: Any, xs: List[Value])              extends Value
  case class ClosV(expr: Expr, var env: Environment)    extends Value
}
```

Note that the `env` variable is a `var`, since we need to modify this in order to fully support `Letrec` expressions with the correct environments (Waddell et al., 2005). This does make the interpretation function not entirely pure, but this is done to implement `Letrec` expressions more easily. It should be noted that the language itself is pure, since a pure interpretation method can be implemented using Y-combinators (Park, 1976). The `Environment` class, which acts as a placeholder for a `Map` instance. For the sake of simplicity, we define this `Map` instance to map variable names to their corresponding values. The map instance would have the type `Map[String, Value]`. The `Environment` class should have a method to `get` and `set` values. This can be achieved by implementing it in the following way:

```
case class Environment(maps: Map[String, Value]) {
  def set(t: (String, Value)): Environment = Environment(maps + t)

  def get(name: String): Value = maps(name)
}
```

At the start of interpreting an expression, we do not have an `Environment`, so we define the environment to be standardized to `Environment(Map())`. Using this as the backbone for running programs, we can define a simple implementation for a basic interpreter:

```scala
object Interpreter {
  def interp(e: Expr, env: Environment = Environment(Map())): Value = e match {
    case Var(name) => env.get(name)

    case Val(value, args) => ValV(value, args.map(x => force(interp(x, env))))

    case Letrec(bindings, body) =>
      val env1 =
        bindings.foldLeft(env)((e, a) => e.set(a._1.name -> ClosV(a._2, null)))
      bindings.foreach(b => env1.get(b._1.name).asInstanceOf[ClosV].env = env1)
      interp(body, env1)

    case Function(arg, body) => ClosV(Function(arg, body), env)

    case Match(arg, cases) =>
      val value = force(interp(arg, env))
      val result =
        cases
          .toStream
          .map(c => (c.body, doMatch(value, c.pattern, env)))
          .find(c => c._2 != null)
          .getOrElse(throw new Exception("MatchException"))

      interp(result._1, result._2)

    case Apply(b, arg) => interp(b, env) match {
      case ClosV(Function(param, body), closureEnv) =>
        val env1 = closureEnv.set(
          param.asInstanceOf[Var].name -> interp(arg, env)
        )
        interp(body, env1)
    }
  }
```

Figure 1: A basic implementation of the interpreter

The `force` method forces interpretation on `ClosV` values. Since `ClosV` values are unevaluated expressions, where the environment is already given, we only need to force the interpretation of the delayed result. This is done by recursively evaluating the expression with the stored environment, until a normal value is returned.

The interpretation of `Letrec` expressions is a bit tricky, since the functions should be able to call themselves. In order to solve this, we set the value of the function as `ClosV(..., null)`, indicating that the environment is uninitialized. We collect all these mappings and store them in a new environment, and eventually initialize the environment of each of these `ClosV`s to this new environment. Interpretation of `Function`s results in a `ClosV` constructed with the current environment.

Application of these functions, using `Apply`, is interpreted by interpreting the body of the `Apply` expression. Since it is only possible to interpret `Function` expressions, the interpretation of the body should always result in a `ClosV(Function(param, body), closureEnv)`. We

can then interpret the body of the function using the resulting `closureEnv`.

Lastly, the implementation of the pattern matching feature is done with the `Match` expression. Matching is done using the `doMatch`, which is implemented in the following way:

```
def doMatch(left: Value, right: Pattern, env: Environment): Environment = {
  // When the environment equals null, this indicates that a match is
  // not possible, and therefore we immediately return null.
  if (env == null)
    return null

  (left, right) match {
    case (ValV(l, ls), ValP(r, rs)) if l.equals(r) =>
      ls.zip(rs).foldLeft(env)((e, t) => doMatch(t._1, t._2, e))
    case (v @ ValV(_, _), VarP(name)) => env.set(name -> v)
    case _ => null
  }
}
```

Figure 2: The matching algorithm for the basic interpreter

All this together gives us a basic interpreter that handles pattern matching. The full code can be found at Appendix A. Using this interpreter, we can interpret programs using the pattern matching feature. An example of a program that implements the `append` function can be found at Appendix B or here for the full version.

2.3 Abstract Interpretation

Symbolic analysis refers to the analysis of programs and determines what symbolic values will cause which part of the program to execute. Rather than actually executing the interpreter, it assumes symbolic values for variables and follows the path that will be executed. This technique is also known as *abstract interpretation* (Cousot and Cousot, 1977). In order to generate test data, we would need to perform such abstract interpretation of a program. Running such an interpreter on a program would need to result in some kind of abstract representation of the sequence of computations. A way of doing this is possible by using *free monads* (Trnková et al., 1975). This allows us to transform a normal interpreter into multiple different interpreters, by redefining the meaning of the commands of the interpreter.

This makes it possible to transform a simple deterministic interpreter into an abstract interpreter, allowing us to gain information about its semantics, without actually performing the computations. In the next section, we will explain how we can redefine regular interpreters using free monads and what strategies we can use in order to handle nondeterminism.

3 Transformation into a Symbolic Executor

In this section, we will discuss how we can transform a regular interpreter into an abstract interpreter that performs symbolic execution, using free monads. The first subsection will give an introduction to what free monads are and why we need them for symbolic execution. Furthermore, we will explain how a program represented as a free monad can be executed and what execution strategies we can use, in order to get the concrete values of the interpretation.

3.1 Free Monads

A monad is a design pattern that has two functions, `bind` and `return`. The `return` function simply turns a value into a monadic value, while the `bind` function maps takes a function and maps the function on the monad. This makes it possible to chain computations in some specific and useful way.

The free monad (Trnková et al., 1975) is a design pattern that obeys the laws of a monad, and therefore has a `bind` and `return` function. The difference between the free monad and other monads is that it does not actually perform the computations. Chaining operations just builds up a nested series of commands, resulting in a command tree. This provides a flexible way of defining the *actual* meaning of such a composition and how to handle a free monads. In this paper, we implement the free monad using the following classic inductive definition:

```
sealed trait Free[F[_], A] {
  def bind[B](f: A => Free[F, B]): Free[F, B] = Bind(this, f)
}

case class Pure[F[_], A](a: A) extends Free[F, A]
case class Bind[F[_], X, A](x: Free[F, X], f: X => Free[F, A]) extends Free[F, A]
```

The `Pure` case class represents the `return` method of the monad, which accepts a pure value and wraps it into a free monad. Computation of `Pure` instances returns a value of type `A`. The `Bind` case class represent the `bind` method (also known as `flatMap`), which takes a free monad and a continuation function. All this together creates the basic foundation of the free monad.

In addition to these case classes, we also need to have some kind of way to represent multiple `Free[F[_], A]` instances, as we need to be able to represent branches contain all possible choices in a traversable way. Therefore, we extend the free monad to have a `Fork` case class:

```
case class Fork[F[_], A](fa: () => Stream[Free[F, A]]) extends Free[F, A]
```

This addition turns the free monad into a free `MonadPlus`, which we need for the search of solution of non-deterministic computations (Reck and Fischer, 2009). In this case, the `fa` property is a nullary lambda rather than regular `Stream`, as the head of the stream is strict, while we do not want to compute this value immediately.

The free monad has a `bind` method, which binds a continuation function to the free monad, using the `Bind` case class. In order to create symbolic executors, we need to create an interface for these executors, commonly referred as the natural transformation function. In Scala, these free monads are typically used for programs in the form of for-comprehensions, but since our programs are expressions, our natural transformation interface is a bit different. The pipeline of how programs are transformed and executed is as following:

```
F[A] --transform--> Free[F, A] --handleFree--> G[A]
```

Using this pipeline, we can create the actual interface that we will need to implement:

```
object Free {
  trait ~>[F[_], G[_]] {

    def monad: MonadPlus[G]
    def transform[A](fa: F[A]): Free[F, A]
    def handle[A](fa: F[A]): G[A] = handleFree(transform(fa))
    def handleFree[A](free: Free[F, A]): G[A] = free match {
      case Pure(x) => monad.pure(x)
      case Bind(x, f) =>
        monad.bind(handleFree(x),
                   (a: Any) => handleFree(f.asInstanceOf[Any => Free[F, A]](a)))
      case Fork(fa) =>
        fa.apply()
          .foldLeft(monad.mzero: G[A])((b, f) => monad.mplus(b, handleFree(f)))
  }
}
```

Figure 3: The natural transformation interface

As we can see, we only need to implement the `transform` method and define the operations for the `MonadPlus`. The `MonadPlus` is an extension of the `Monad`, and are both defined in the following way:

```
trait Monad[M[_]] {
  def pure[A](x: A): M[A]
  def bind[A, B](ma: M[A], f: A => M[B]): M[B]
}

trait MonadPlus[M[_]] extends Monad[M] {
  def mzero[A]: M[A]
  def mplus[A](m1: M[A], m2: M[A]): M[A]
}
```

Figure 4: The interfaces for the `Monad` and the `MonadPlus`

Using these interfaces, we can implement a symbolic executor for a language, such that the interpreter is able to handle non-deterministic choices and logic variables. We can even extend our language with new commands, which we will do in order to generate test data.

3.2 Introducing Logic Expressions

In order to make our basic interpreter an interpreter that handles functional programming, we need to add logic expressions and values, which we then can use to solve logical equations. We can use these logic expression to define methods that define the domain of a variable. This particular operator is called the *amb* operator, short for the ambiguity operator (McCarthy, 1959). The `amb`-operator is an operator that expresses non-determinism. The `amb`-operator basically defines a domain for a certain variable. In our language, if we allow logic variables without any value assigned to them, we can implement the `amb`-operator as follows:

```
(Var("amb"),
    Function(Var("left"),
        Function(Var("right"), Match(Var("dummy"), List(
            Case(ValP("dummy1", Nil), Var("left")),
            Case(ValP("dummy2", Nil), Var("right"))
        )))
    )
)
```

Figure 5: A hypothetical implementation of the ambiguity operator

Assuming that `Var("dummy")` is not set, the function can either return `Var("left")` or `Var("right")`, since the dummy variable is able to be matched with both values, and makes the result of the function ambiguous. Interpreting this with the basic interpreter does not work, since `Var("dummy")` is a free variable. Therefore, we need to find a way to allow free variables. This is done by adding `MetaV` as a `Value` case class, which represent meta variables or logic variables. In order to define these logic variables, we add a new expression `WithFree`. This binds the name of the variable to the meta value. Such an expression is defined as:

```
case class WithFree(v: Var, body: Expr[Value]) extends Expr[Value]
```

For example, using this expression as `WithFree(Var("x"), <expr>)` is equivalent in the Curry language to `<expr> where x free`.

3.3 Evaluation into a Command Tree

In order to implement a symbolic executor, we need to implement the trait `F ~> R` (see Figure 3), where `F` is a functor and `R` is a type container indicating the result from running the transformed programs. For example, the type container can be:

- `Id[A]`, which is the `Id` monad. This indicates that the execution of a program will result in just `A`.
- `Future[A]` is used for asynchronous computations.
- `Stream[A]` is used for gathering multiple results.
- `Option[A]` can be used for optional results.
- etc.

In this case, we want to define an interpreter that generates a (possibly infinite) stream of values. The functor in this case is `Expr` and the type container is a `Stream`. We define a symbolic executor by extending this trait:

```
object SymbolicInterp extends (Expr ~> Stream) {
  implicit def value2free[F[_], A](value: Value): Free[F, A] =
    Pure[F, A](value.asInstanceOf[A])
  override def monad: MonadPlus[Stream] = new MonadPlus[Stream] {
    override def mzero[A]: Stream[A] = Stream()
    override def mplus[A](m1: Stream[A], m2: Stream[A]): Stream[A] = m1 ++ m2
    override def pure[A](x: A): Stream[A] = Stream(x)
    override def bind[A, B](ma: Stream[A], f: A => Stream[B]): Stream[B] =
      ma.flatMap(f)
  }

  override def transform[A](expr: Expr[A]): Free[Expr, A] = interp(expr)

  def interp[A](expr: Expr[A]): Free[Expr, A] = interp(expr, Environment(Map()))
  def interp[A](expr: Expr[A], env: Environment): Free[Expr, A] = expr match {
    case Var(name) => env.get(name)
    case Val(v, vs) => ValV(v, vs.map(x => handleFree(interp(x, env)).head))
    case Function(arg, body) => ClosV(Function(arg, body), env)
    case Letrec(bindings, body) =>
      val env1 =
        bindings.foldLeft(env)((e, a) => e.set(a._1.name -> ClosV(a._2, null)))
      bindings.foreach(b => env1.get(b._1.name).asInstanceOf[ClosV].env = env1)
      interp(body, env1).asInstanceOf[Free[Expr, A]]

    case Exists(v, cond) =>
      val env1 = env.set(v.name -> MetaV(v.name))
      cond match {
        case Equals(left, right) =>
          interp(left, env1).bind(r1 =>
            interp(right, env1).bind(r2 =>
              Pure(directUnify(r1, r2, env1)))).bind(e => e.get(v.name))
      }

    case WithFree(Var(name), body) =>
      interp(body, env.set(name -> MetaV(name))).asInstanceOf[Free[Expr, A]]

    case Match(arg, cases) =>
      interp(arg, env).bind(r1 => Fork(() =>
        cases
          .toStream
          .map(c => (c.body, doMatch(r1, c.pattern, env)))
          .filter(c => c._2 != null)
          .map(c => interp(c._1, c._2).asInstanceOf[Free[Expr, A]])
      ))
```

```
    case Apply(e, arg) => interp(e, env).bind {
      case ClosV(Function(param, body), closureEnv) =>
        interp(arg, env).bind(r1 => {
          val env1 = closureEnv.set(param.name -> r1)
          interp(body, env1)
        }.asInstanceOf[Free[Expr, A]])
    }
  }
```

Figure 6: An implementation of the symbolic executor

As we can see, we implemented the `monad` method, which implements the `MonadPlus` interface for a `Stream`. The `mzero` method returns an empty instance, while `mplus` concatenates two instances. The `pure` method takes an instance of `A` and returns a `Stream[A]` which contains only one item, and the `bind` method is the same as the `flatMap` method, which is trivial to implement for streams.

We have added an *implicit* `value2free` method, which implicitly converts `Value` instances into instances of `Free[F, A]`, which is simply done by using the `Pure` constructor. The `transform` method is performed using the `interp` function, which results in a free monad tree. As we can see, we need to make some slight changes compared to the basic interpreter. Some of the expressions are interpreted in the same way, like `Var`, `Function` and `Letrec`. Expressions that are interpreted with inner interpretations use the `bind` method to use the resulting values (such as the `Apply` expression).

The `WithFree` expression is interpreted by mapping the name of the given variable to a meta variable, indicating that this is a logic variable in the scope of the body. Using this expression, we can now define a working implementation of the ambiguity operator (shown in Figure 5):

```
(Var("amb"),
  Function(Var("left"), Function(Var("right"),
    WithFree(Var("dummy"), Match(Var("dummy"), List(
      Case(ValP("dummy1", Nil), Var("left")),
      Case(ValP("dummy2", Nil), Var("right"))
))))))
```

Figure 7: An implementation of the ambiguity operator with a free variable

This function allows us to specify non-deterministic choices. However, one of the big changes from the basic interpreter to the symbolic executor is the `Match` expression. Since meta variables are now valid `Value` instances, we need to change the `doMatch` method (see Figure 2), such that it can handle `MetaV` instances as well. Therefore, we need to add the following case to the `doMatch` method:

```
case (MetaV(name), ValP(r, rs)) =>
  val rm = rs.map(_ => fresh())
  val env1 = rm.foldLeft(env)((e, r) => e.set(r.name -> r))
  rm.zip(rs)
    .foldLeft(env1.set(name -> ValV(r, rm)))((e, t) => doMatch(t._1, t._2, e))
```

This returns an environment where the meta variable is replaced by a (meta) value, where the list of remaining arguments (`rm`) are fresh variables (instantiated using the `fresh` method). The `fresh` method simply returns a `MetaV` instance with a name that has not been used before.

The `Match` expression is also different compared to the other expressions, as this is the only expression where non-deterministic choices can happen. Therefore, the `Match` expression is the only expression that returns a `Fork` instance, containing all the interpretations of the cases where the value was able to be matched to the pattern. Currently, this will be searched through (with the `handleFree` method) in a depth-first manner, but is easily adjustable by overriding the `handleFree` method.

The last thing that we also need in order to generate test data is to add an existential quantifier which can accept logical expressions (expressions that represent boolean constraints). This allows us to write queries in a declarative manner, which will result in solutions that we can use as test data. We do this by extending the `Expr[R]` trait as follows:

```
sealed trait LogicExpr[R] extends Expr[R]

object LogicExpr {
  case class Exists(arg: Var, cond: LogicExpr[Value]) extends LogicExpr[Value]
  case class Equals(l: Expr[Value], r: Expr[Value]) extends LogicExpr[Value]
}
```

As we can see at the interpretation of the `Exists` expression (at Figure 6), we intepret both values of the `Equals` constraint and perform *direct unification*. This is done using the `directUnify` method, which acts similarly to the `doMatch` method, but matches two `Values` rather than a `Value` and a `Pattern`. A trivial implementation of this is done as follows:

```
def directUnify(left: Value, right: Value, env: Environment): Environment = {
  if (env == null)
    return null

  (left, right) match {
    case (ValV(l, ls), ValV(r, rs)) if l.equals(r) =>
        ls.zip(rs).foldLeft(env)((e, t) => directUnify(t._1, t._2, e))
    case (MetaV(name), ValV(r, rs)) => env.set(name -> ValV(r, rs))
    case (MetaV(n1), MetaV(n2)) if n1.equals(n2) => env
    case (MetaV(n1), MetaV(n2)) => env.set(n1 -> MetaV(n2)).set(n2 -> MetaV(n1))
    case _ => null
  }
}
```

With this implementation, we are able to write declarative queries which will give us the solution. For example:

```
Letrec(
  List(
    (Var("append"), Function(Var("x"), Function(Var("ys"),
      Match(Var("x"), List(
        Case(ValP("Empty", Nil),
          Var("ys")),
        Case(ValP("Cons", List(VarP("a"), VarP("as"))),
```

```
          Val("Cons", List(
            Var("a"),
            Apply(Apply(Var("append"), Var("as")), Var("ys")))
          ))
      )))),
  ),
  Exists(Var("x"),
    Equals(
      Apply(Apply(
        Var("append"), Val("Cons", List(
          Val("Num", List(Val(1, Nil))),
          Val("Empty", Nil))
        )
      ), Var("x")),
      Apply(Apply(Var("append"), list1), list2)))
)
```

Where `list1` is equal to `[1, 2]` and `list2` is equal to `[3, 4, 5]` (written in the same notation as discussed at Section 2.1). The program is roughly equivalent to the following Prolog-like pseudocode:

```
append [] ys       = ys
append [a | as] ys = [a | append as ys]

exists x ~> append [1] x == append [1, 2] [3, 4, 5]
```

When running, this program, the result turns out to be a stream containing a single element, which is:

```
ValV("Cons",
    List(ValV("Num", List(ValV(2, List()))),
        ValV("Cons", List(ValV("Num", List(ValV(3, List()))),
            ValV("Cons", List(ValV("Num", List(ValV(4, List()))),
                ValV("Cons", List(ValV("Num", List(ValV(5, List()))),
                ValV("Empty", List()))))))))))
```

Which roughly translates that `x == [2, 3, 4, 5]` is the only solution possible for the program. Using this, we can define our own abstract data types and functions, on which we run these kinds of queries to generate test cases.

3.4 Handling the Free monad

Using the basic definition of the natural transformation interface, we are able to write existential queries, but the symbolic analysis is still done in a depth-first way. Luckily, the free monad provides us the flexibility to do the analysis in any way we want, for example breadth-first rather than depth-first search. This way, we would not end up stuck in some infinite recurion. For example, the following program given by Tolmach and Antoy (2003), shows that a depth-first search would never terminate, while a breadth-first search will end up in an infinite stream of `1`s:

```
Letrec(
  List(
    (Var("f"),
      Function(Var("x"),
        WithFree(Var("dummy"), Match(Var("dummy"), List(
          Case(ValP("dummy1", Nil), Apply(Var("f"), Var("x"))),
          Case(ValP("dummy2", Nil), Var("x"))
        ))))),
  ),

  Apply(Var("f"), Num(1))
)
```

Running this program with the interpreter we currently have, will end up in an infinite recursion. In order to solve this problem, we manually override the `handleFree` method, such that it will perform a breadth-first search on the free monad:

```
override def handleFree[A](free: Free[Expr, A]): Stream[A] = bfs(Stream(free))

def bfs[A](free: Stream[Free[Expr, A]]): Stream[A] = {
  free match {
    case Pure(x) #:: xs => x #:: bfs(xs)
    case Bind(x, f) #:: xs => bfs(xs) #:::
        monad.bind(handleFree(x),
                   (a: Any) => handleFree(f.asInstanceOf[Any => Free[Expr, A]](a)))
    case Fork(fa) #:: xs => bfs(xs #::: fa.apply())
    case Empty => Stream()
  }
}
```

We can now access the stream by running `SymbolicInterp.handle(code).head`, which will indeed result in `ValV("Num", List(ValV(1, List())))`. This proves that we are actually able to perform different search strategies on the free monad, due to its flexibility and structure. This could be improved even further by performing some kind of search with optimization heuristics.

4 Conclusion and Future Work

We have shown how to turn a basic interpreter for a language that supports pattern-matching into a symbolically executable interpreter. This shows that the depth-first search that symbolic execution algorithms commonly use can be replaced by other search strategies, such as breadth-first search. Furthermore, we have seen how to create a custom language based upon the basic interpreter, such that it can handle non-deterministic choices.

The usage of free monads for constructing these abstract command trees allows us to alter the 'meaning' of the command tree in a flexible way. In Section 3.4, we have seen that traversing the free monad can be adjusted if a different kind of traversal method is preferred. Furthermore, the free monad gives us many flexible options for how programs will be interpreted and how commands or expressions can be changed.

One thing that might be improved is the refactoring the `Fork` case class as a command, rather than a constructor for the free monad. The addition of the `Fork` makes the free monad not an actual free monad, but more like a hybrid between the free monad and the tree monad. Such a refactor would make the definition of the free monad more conceptually cleaner, and makes implementing symbolic executors better understandable.

Also, support for other constraints, such as `Or` and `And` could perhaps be added, to create more directed and specific queries. Perhaps some kind of SMT solver could be used for this, such as Z3 (De Moura and Bjørner, 2008). This would also add the possibility to extend the symbolic executor to solve more intricate queries, e.g. supporting linear arithmetic.

References

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224.

Claessen, K. and Hughes, J. (2011). Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320.

McCarthy, J. (1959). A basis for a mathematical theory of computation. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 33–70. Elsevier.

Park, D. M. (1976). The y-combinator in scott's lambda-calculus models.

Reck, F. and Fischer, S. (2009). Towards a parallel search for solutions of non-deterministic computations. *GI Jahrestagung*, 154:2889–2900.

Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Sen, K. (2007). Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM.

Sussman, G. J. and Steele, G. L. (1998). Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439.

Tolmach, A. and Antoy, S. (2003). A monadic semantics for core curry. *Electronic Notes in Theoretical Computer Science*, 86(3):16–34.

Trnková, V., Adámek, J., Koubek, V., and Reiterman, J. (1975). Free algebras, input processes and free monads. *Commentationes Mathematicae Universitatis Carolinae*, 16(2):339–351.

Waddell, O., Sarkar, D., and Dybvig, R. K. (2005). Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher-Order and Symbolic Computation*, 18(3-4):299–326.

Appendices

A Basic Interpreter for Pattern Matching

```
import Expr._
import Pattern._
import Value._

trait Expr

object Expr {
  case class Var(name: String) extends Expr
  case class Val(value: Any, args: List[Expr]) extends Expr
  case class Function(arg: Var, body: Expr) extends Expr
  case class Letrec(bindings: List[(Var, Function)], body: Expr) extends Expr
  case class Apply(body: Expr, arg: Expr) extends Expr
  case class Case(pattern: Pattern, body: Expr) extends Expr
  case class Match(arg: Expr, cases: List[Case]) extends Expr
}

sealed trait Pattern

object Pattern {
  case class ValP(value: Any, args: List[Pattern]) extends Pattern
  case class VarP(name: String) extends Pattern
}

sealed trait Value

object Value {
  case class ValV(x: Any, xs: List[Value]) extends Value
  case class ClosV(expr: Expr, var env: Environment) extends Value
}

case class Environment(maps: Map[String, Value]) {
  def set(t: (String, Value)): Environment = Environment(maps + t)

  def get(name: String): Value = maps(name)
}

object Interpreter {

  def interp(e: Expr, env: Environment = Environment(Map())): Value = e match {
    case Var(name) => env.get(name)
    case Val(value, args) => ValV(value, args.map(x => force(interp(x, env))))

    case Letrec(bindings, body) =>
```

```
      val env1 =
        bindings.foldLeft(env)((e, a) => e.set(a._1.name -> ClosV(a._2, null)))
      bindings.foreach(b => env1.get(b._1.name).asInstanceOf[ClosV].env = env1)
      interp(body, env1)

    case Function(arg, body) => ClosV(Function(arg, body), env)

    case Match(arg, cases) =>
      val value = force(interp(arg, env))
      val result =
        cases
          .toStream
          .map(c => (c.body, doMatch(value, c.pattern, env)))
          .find(c => c._2 != null)
          .getOrElse(throw new Exception("MatchException"))

      interp(result._1, result._2)

    case Apply(b, arg) => interp(b, env) match {
      case ClosV(Function(param, body), closureEnv) =>
        val env1 = closureEnv.set(
          param.asInstanceOf[Var].name -> interp(arg, env)
        )
        interp(body, env1)
    }
  }

  def force(value: Value): Value = value match {
    case ClosV(expr, env) => force(interp(expr, env))
    case v => v
  }

  def doMatch(left: Value, right: Pattern, env: Environment): Environment = {
    if (env == null)
      return null

    (left, right) match {
      case (ValV(l, ls), ValP(r, rs)) if l.equals(r) =>
        ls.zip(rs).foldLeft(env)((e, t) => doMatch(t._1, t._2, e))
      case (v @ ValV(_, _), VarP(name)) => env.set(name -> v)
      case _ => null
    }
  }
}
```

B Basic Program Example

```
object App {
  def main(args: Array[String]): Unit = {
    val list1 =
      Val("Cons", List(Val("Num", List(Val(1, List()))),
        Val("Cons", List(Val("Num", List(Val(2, List()))),
          Val("Empty", List())))))
    val list2 =
      Val("Cons", List(Val("Num", List(Val(3, List()))),
        Val("Cons", List(Val("Num", List(Val(4, List()))),
          Val("Cons", List(Val("Num", List(Val(5, List()))),
            Val("Empty", List()))))))))

    val code = Letrec(
      List(
        (Var("append"),
          Function(Var("x"), Function(Var("ys"), Match(Var("x"), List(
            Case(ValP("Empty", Nil),
              Var("ys")),

            Case(ValP("Cons", List(VarP("a"), VarP("as"))),
              Val("Cons", List(Var("a"),
                Apply(Apply(Var("append"), Var("as")), Var("ys")))))
        )))))
      ),
      Apply(Apply(Var("append"), list1), list2)
    )

    println(formalized(Interpreter.interp(code)))
  }

  def formalized(value: Value): String = value match {
    case ValV(x, Nil) => x.toString
    case ValV(x, xs) => s"$x(${xs.map(formalized).mkString(", ")})"
    case x => x.toString
  }
}
```