Semantics Exploration: Surprise in Language Design

Eli Rosenthal

May 16, 2016

Overview

When are Languages Surprising?

- A lot of programming languages have similar constructs. For various reasons, programmers build up certain models of how these constructs behave.
- *Personal Example:* I expect lexical scoping when I encounter a feature in a new language described as a "lambda function."
  - Possibly because my first language was an HtDP teaching language, followed closely by ML.
  - Also potentially because I gravitate to static scoping disciplines because they are more compositional than dynamic scope.
- Another common example is the extent to which standard arithmetic operators are not commutative or even associative in many popular programming languages, be it due to floating point, or "OO"+ operator overloading.

When are Languages Surprising?

- A lot of programming languages have similar constructs. For various reasons, programmers build up certain models of how these constructs behave.
- *Personal Example:* I expect lexical scoping when I encounter a feature in a new language described as a "lambda function."
  - Possibly because my first language was an HtDP teaching language, followed closely by ML.
  - Also potentially because I gravitate to static scoping disciplines because they are more compositional than dynamic scope.
- Another common example is the extent to which standard arithmetic operators are not commutative or even associative in many popular programming languages, be it due to floating point, or "OO"+ operator overloading.

When are Languages Surprising?

- A lot of programming languages have similar constructs. For various reasons, programmers build up certain models of how these constructs behave.
- *Personal Example:* I expect lexical scoping when I encounter a feature in a new language described as a "lambda function."
  - Possibly because my first language was an HtDP teaching language, followed closely by ML.
  - Also potentially because I gravitate to static scoping disciplines because they are more compositional than dynamic scope.
- Another common example is the extent to which standard arithmetic operators are not commutative or even associative in many popular programming languages, be it due to floating point, or "OO"+ operator overloading.

Background/Related Work

I wasn't sure there was any... but then Shriram told me to look at this paper

- Felleisen's 1990 paper "On the Expressive Power of Programming Languages." Provides a formal account of how expressive a language is based on operational semantics, as opposed to computability.
- Define a language $\mathcal{L}$ as a set of freely generated *phrases* from a grammar, a subset of valid programs, a set of values, and an *operational semantics*
- The above definition gives rise to a natural notion of *sublanguage*, and *syntactic abstraction*
- This provides a formal foundation for how expressive a language is, based on what features can be expressed in terms of some desugaring (skipping over many details here) that preserves behavioral equivalence.
- This set of definitions also (albiet implicitly) lays out conditions for features to be implemented orthogonally,

Roadmap

- Surprise
  - Formal/idealized Definition
  - Practical implementation/tooling
- Orthogonality (How might we avoid surprises?)
  - Writing interpreters as *compositions of features* that are orthogonal by costruction (and hence compositional).

Overview

Surprise: the Ideal case

Consider two langauges, R, and the λ calculus.

- In an idealized setting, assume these languages have some notion of syntax, and an operational semantics.
- Ask the question of "to what extent is R's scoping surprising?"
  - This implies a syntactic mapping $\mathcal{E}$ of the sort

    $$\mathcal{E}(\texttt{function(a,b)d}) = \lambda a\, b.\, \mathcal{E}(d)$$

    Along with some natural inverse mapping. Note that this mapping can even be partial, implicitly introducing stuck states in λ; these act as uninterpreted values.
  - If we can extend this syntactic mapping to a mapping of *evaluation contexts*, then we can create a hybrid operational semantics that applies this mapping, takes some number of λ evaluation steps, and maps it back to R.
- If this new composite relation is nondeterministic, then there is some sequence of λ-calculus reduction steps that breaks with R's semantics.

Limitations

- I think this is actually a nice definition, and seems to fit into a natural extension into Felleisen's framework. It's really Felleisen+re-sugaring, allowing us to consider behavioral equivalence in the host language as opposed to the de-sugared language. Assuming I'm understanding the paper right
- But it significantly limits what languages we can model; we would need an operational semantics for R, which is not easy

Idea Limit ourselves to syntactic mappings + evaluator for the big language, see how far we can get.

Limitations

- I think this is actually a nice definition, and seems to fit into a natural extension into Felleisen's framework. It's really Felleisen+re-sugaring, allowing us to consider behavioral equivalence in the host language as opposed to the de-sugared language. Assuming I'm understanding the paper right
- But it significantly limits what languages we can model; we would need an operational semantics for R, which is not easy

Idea Limit ourselves to syntactic mappings + evaluator for the big language, see how far we can get.

Limitations

- I think this is actually a nice definition, and seems to fit into a natural extension into Felleisen's framework. It's really Felleisen+re-sugaring, allowing us to consider behavioral equivalence in the host language as opposed to the de-sugared language. Assuming I'm understanding the paper right
- But it significantly limits what languages we can model; we would need an operational semantics for R, which is not easy

Idea Limit ourselves to syntactic mappings + evaluator for the big language, see how far we can get.

Algebraic Data-types and Recursion

We will operate on *unrolled* algebraic data-types (ADTs), i.e. given a standard recursive ADT

```
data Exp = Number Integer | Add Exp Exp
```

We expect it to be given as

```
data Exp a = Number Integer | Add a a
```

Where we can recover a type isomorphic to the original `Exp` by tying the recursive knot ourselves:

```
newtype Mu f = Mu (f (Mu f))
type Exp' = Exp (Mu Exp)
```

This grants us greater flexibility.

Interleaving Two ADTs

This representation allows us to account for two ADTS where each term allows sub-terms of either type.

```
newtype CR a b = CR (Either (a (CR a b)) (b (CR a b)))
type a :+: b = CR a b
```

This lets us define easy syntactic mappings and fuzzing operations.
(Look at Code)

Testing for violations

- Given this mapping, we can subsitute terms in for referentially transparent ones.
- With the λ-calculus, this is some number of β/η rule applications.
- This, along with `Arbitrary` instances, is fertile ground for automated counterexample finding!

(Look at Code)

Overview

Avoiding Surprise

- A related notion to surprise is that of *orthogonality*, which (I think) is the idea that different language features do not interact in destructive ways.
- This implies some notion of composition: if a given features does not interfere with others, than evaluation should be the structural thing that we expect.

Idea Reverse-engineer the syntactic approach from before to create features that are orthogonal by construction.

Consider this simple language:

```
data Exp
  = Id String
  | Lam String Exp
  | App Exp Exp
  | ArithOp (Integer -> Integer -> Integer) Exp Exp
  | N Integer
  | BoolOp (Bool -> Bool -> Bool) Exp Exp
  | B Bool
  | IF Exp Exp Exp
```

Then refactor it into its component features, and unwrapped:

```
data UTExp a  = Id String | Lam String a | App a
data ArithExp a
    = N Integer
    | ArithOp (Integer -> Integer -> Integer) a
data BoolExp a
    = B Boolean
    | BoolOp (Boolean -> Boolean -> Boolean) a
data Exp = AE ArithExp | BE BoolExp | UE UTExp

-- value type!
data Val = NV Integer | BV Boolean | Clos Exp
```

Then write an evaluator with the "extensible let-rec trick"

```
arithEval _    (N n)              = Just (NV n)
arithEval eval (ArithOp f a b) = do
  a' <- eval a
  b' <- eval b
  case (a', b') of
     (Just (NV x), Just (NV y)) -> return $ NV $ f x y
     _ -> Nothing

boolEval :: (a -> Maybe Val) -> BoolExp a -> Maybe Val
-- ...

utlcEval :: (a -> Maybe Val) -> UTExp a -> Maybe Val
-- ...

eval :: Exp -> Maybe val
eval (AE a) = arithEval eval a
eval (BE a) = boolEVal eval a
eval (UE a) = utlcEval eval a
```

What's the Point?

- Parametricity
  - The evaluators for individual features cannot even inspect what features are in the languages.
  - Orthogonality appears to be a free theorem!
- Though there are some scaling issues, a lot of this can be automated.
- Also elucidates "copmosition" bent to earlier part of project. Note that

$$\texttt{Exp} \simeq \texttt{ArithExp :+: UTExp :+: BoolExp}$$

If some compositional language models the features of a larger one, then it effectively factors out into a "bag-of-features" data type, and quick-check is what ensures parametricity.

What's the Point?

- Parametricity
  - The evaluators for individual features cannot even inspect what features are in the languages.
  - Orthogonality appears to be a free theorem!
- Though there are some scaling issues, a lot of this can be automated.
- Also elucidates "copmosition" bent to earlier part of project. Note that

$$\texttt{Exp} \simeq \texttt{ArithExp :+: UTExp :+: BoolExp}$$

If some compositional language models the features of a larger one, then it effectively factors out into a "bag-of-features" data type, and quick-check is what ensures parametricity.

What's the Point?

- Parametricity
  - The evaluators for individual features cannot even inspect what features are in the languages.
  - Orthogonality appears to be a free theorem!
- Though there are some scaling issues, a lot of this can be automated.
- Also elucidates "copmosition" bent to earlier part of project. Note that

$$\texttt{Exp} \simeq \texttt{ArithExp :+: UTExp :+: BoolExp}$$

  If some compositional language models the features of a larger one, then it effectively factors out into a "bag-of-features" data type, and quick-check is what ensures parametricity.

That's about it! Questions?