Abstract

We define a notion of *surprise* between two programming languages which provides a formal analog to the notion that “A user experienced with Scheme is surprised by feature x of Javascript”. Along the way we give clear definitions of what it means for a language feature to be *orthogonal*, as well as provide an automated tool for exploring (in a purely syntactic manner) the extent to which a language implements a feature in an unusual way.

1 Introduction: Two Parables

We begin considering an idealized version of two common cases of “programming language aquisition.”

1.1 The Intro CS Student

Consider a student X taking an introductory programming course (CS-A). X has no prior programming experience, and their first assignment is to use their language L_0 as a calculator. L_0 is a small “expression language” with numbers and basic arithmetic operations.

```
;; Assignment 1
; Compute $\sqrt{157482506564341206945618310642111308294618078190384348521}$
(sqrt 157482506564341206945618310642111308294618078190384348521)
; => 12549203423498290193999432211
```

In their next assignment, X now has access to conditional branching and functions. To use these features, X writes their assignment in L_1, which also has all of the features of L_0.

```
;; Assignment 2
; Write a function which given two numbers $x, y$ computes $\sqrt{xy}$
(define (f x y) (sqrt (* x y)))
```

While X views this assignment as building on previous techniques, they don’t realize that the semantics of `sqrt` and `*` have changed! In L_0, they were special forms built into the language, and in L_1 they are implemented as functions in the standard library. X may not have noticed this because there are few *observations* that they could make on language output that could distinguish these two scenarios.

The following semester, X’s instructors have decided they have to learn a “practical language” because X and their classmates have to be more attractive to employers, or some other bullshit reason; X’s class (CS-B) is hence taught in python! Python has different syntax than the L_i that X learned in CS-A, so X does some of the CS-A assignments and tries to translate them to python.

```
# Assignment 1
from math import sqrt
sqrt(157482506564341206945618310642111308294618078190384348521)
# => 1.254920342349829e+28
# Assignment 2
def f(x):
    return sqrt(x * y)
```

Both of these examples surprised X! In assignment 1, the first answer is wrong (python's `math.sqrt` does not provide an exact answer). In assignment 2, X made a mistake. However, X only realized their mistake after running a test case, while $L_{i\geq 1}$ produced a static ("compile-time") error if there are free variables in a function body[1].

1.2 The Programming Languages Expert

Whenever a new language begins to gain traction, it is common for a PL theorist to play around with pathological examples of programs that behave "unexpectedly."[2] This is often a rather useful exercise: it can help programmers avoid hard-to-diagnose bugs[3], and it is often a first step to developing a practical or *tested semantics* for a language without a formal specification. Such specifications, in turn, can lead to type systems and static analyses for the language (hence catching more bugs)[4].

1.3 When are Languages Surprising?

We observe that the two instances cases considered above are both instances of the same notion of surprise: a programmer has some underlying model of how evaluation in the language ought to proceed, but the language does something else. In the case of the PL theorist, the language isn't L_1 or L_0[5]; it may be some variant of the λ-calculus. Moreover, as the first parable alludes to, it is possible to have languages with respect to which some other languages are *not* surprising. L_1, while it had different semantics for L_0's features, was a more comfortable next step than python. Another way of stating this is that the features that L_1 added to L_0 are *orthogonal* to the preexisting features.

2 A Semantic Definition of Surprise

In this section we identify a language $\mathcal{L}$ with its *reduction semantics*: i.e. the tuple $(\mathcal{R}_\mathcal{L}, \mathbb{E}_\mathcal{L}, \rightsquigarrow_\mathcal{L}, \Rightarrow_\mathcal{L})$ denoting the set of a set of redexes, evaluation contexts, reduction relation, and transition relation, respectively.[6]

[1]We note that the Full Monty paper exhibits many more python features that are surprising in this sense. A seasoned schemer may be surprised by some edge-cases in python generators that make them hard to interpret through a straight-forward CPS transform (?).

[2]There is a variant of this parable in which someone who is experienced using a language is surprised by some edge-case in the language: demonstrating that their mental model of the language was not fully in line with the implementation. We can call these variants "Wat talk" variants

[3]Shriram Krishnamurthy's Google+ posts with #swiftlang are probably in this category.

[4]Joe's Javascript types paper was used in Facebook's Flow, right?

[5]Mainly because these languages don't exist, though they bare some resemblance with the HtDP teaching languages.

[6]Using the terminology of *Design Concepts in Programming Languages*

We say $\mathcal{L}'$ is a *sublanguage* of $\mathcal{L}$ (denoted $\mathcal{L}' \subseteq \mathcal{L}$) if each component of $\mathcal{L}'$ is a subset of its corresponding component in $\mathcal{L}$. $\mathcal{L}_1$ *partially models* $\mathcal{L}_2$ if there are functions $m_1 : \mathcal{R}_{\mathcal{L}_1} \to \mathcal{R}_{\mathcal{L}_2}$ and $m_2 : \mathbb{E}_{\mathcal{L}_1} \to \mathbb{E}_{\mathcal{L}_2}$.

Note that the existence of a partial model induces a sublanguage in the language being modeled, given a model $m = (m_1, m_2)$ we can denote such a sublanguage $m(\mathcal{L}_1)$. Given a partial model $m : \mathcal{L}_1 \to \mathcal{L}_2$ we therefore have another language $\mathcal{L}_{1\to 2}$ given by

Note: this is actually an easier definition when we just think of the single reduction relation in a structural operational semantics

(This is the general idea, but something may be messed up here)

$$\begin{aligned}
\mathcal{R}_{\mathcal{L}_{1\to 2}} &= \mathcal{R}_{L_2} \\
\mathbb{E}_{\mathcal{L}_{1\to 2}} &= \mathbb{E}_{\mathcal{L}_2} \\
\rightsquigarrow_{\mathcal{L}_{1\to 2}} &= \{(r, r') \ : \ r \rightsquigarrow_{\mathcal{L}_2} r' \text{ and } r \notin m(\mathcal{L}_1)\} \cup \{(m_1(r), m_1(r')) \ : \ r \rightsquigarrow_{\mathcal{L}_1} r'\} \\
\Rightarrow_{\mathcal{L}_{1\to 2}} &= \{(e\{r\}, e\{r'\}) \ : \ r, r' \in \mathcal{R}_{\mathcal{L}_2}.\ e\{r\} \Rightarrow_{\mathcal{L}_2} e\{r'\} \text{ and } e \notin m(\mathcal{L}_1)\} \\
&\quad \cup \{(m_2(e)\{m_1(r)\}, m_2(e)\{m_1(r')\}) \ : \ r, r' \in m_1(\mathcal{R}_{\mathcal{L}_1})\ e\{r\} \Rightarrow_{\mathcal{L}_1} e\{r'\}\}
\end{aligned}$$

We say that $\mathcal{L}_2$ is *surprising with respect to* $\mathcal{L}_1$ if $\Rightarrow_{\mathcal{L}_{1\to 2}} \cup \Rightarrow_{\mathcal{L}_2}$ yields a non-deterministic[7] relation. $\mathcal{L}_{1\to 2}$ is an alternative semantics for the syntax of $\mathcal{L}_2$; it is the "mental model" that a programmer might have of $\mathcal{L}_2$. This creates surprises when letting $\mathcal{L}_{1\to 2}$ reduce an expression for any number of steps before handing the expression back to the $\mathcal{L}_1$ semantics yields a different result than just evaluating all reduction steps in $\mathcal{L}_1$.[8] If $\mathcal{L}_2$ is not surprising with respect to $\mathcal{L}_1$, then we say language constructs present in $\mathcal{L}_2$ but not in $m(\mathcal{L}_1)$ are *orthogonal* to those in $m(\mathcal{L}_1)$.

3 How this might look

There are two options for how this could look in practice. Say we want to assess the implementation of scoping for a particular language with respect to that of the untyped λ-calculus. There are two ways that I see we could do this. Jack mentioned something about the expressive power of languages as stated in terms of *what you can resugar*. That notion sounds directly applicable to this problem.

3.1 The Easy Way

Take terms that only include UTLC-like features. This would mean very basic functions (depending on the mapping, probably just function application, and maybe assignment), and either evaluating them to completion (in the big-step setting) or replacing them with an equivalent term (in the small-step setting). This is less likely to have strange edge-cases than the option below, but things like free variables are hard in this case, so we still can't

[7] We assume that the semantics were deterministic beforehand; accounting for non-determinism is doable, it's just harder to write down.

[8] TODO: add a helpful diagram

throw errors related to unbound identifiers. The bigger issue is that we narrow the scope of our search, potentially losing out on useful examples of how certain features interact with one another.

Open questions here include whether we can get away with doing this for sub-terms (the answer seems yes for pure languages, and no for anything with state), or if we have to do this for full programs only (in which case it becomes less interesting). We don't have this issue as much in the following approach.

3.2 The Cooler-but-might-not-work way

Allow for the inclusion of terms that we do not understand (arrays, indexing, objects) but map them to *uninterpreted values*. We can then treat these uninterpreted values like free identifiers (not having an LHS for an evaluation relation) and map them back as before. For example, given the following:

```
class Foo(object): ...

def foo(bar):
  x = Foo(bar)
  return x.baz(foobar)
foobar = 3
foo(4)
```

We could map this program to a variant of the λ-calculus with sequencing (useless here, as there is no mutation) and `let` (using scheme syntax here):

```
(do
  (uninterp (class Foo(object): ... ))
  (let ((foo (lambda bar
               (let ((x (Foo bar)))
                 ((. x baz) foobar))))
        (foobar 3))
    (foo 4)))
```

In such a way that we have an inverse that gives us back the same program. I think it's pretty clear how that would work in this case. Evaluating this code to completion should return the same context, but with the `(foo 4)` statement replaced with.

```
(let ((x (Foo bar)))
  ((. x baz) foobar1))
```

Where `foobar1` is a fresh identifier. This code no longer runs to completion in python, so we have discovered something interesting! This also gives us a much simpler algorithm for searching for a surprise, we just evaluate as much of the expression as we can and map it back in.

But there is a problem! What if we had named the class `foobar1`[9]? Then our renaming would cause a previously free identifier to be bound. There doesn't appear to be an easy way to avoid this for languages where different features share a namespace. We could attach asterisks to cases such as these, or we could try to grep the source of the original program for the identifiers we generate. Many possible work-arounds, but it doesn't sound clean. One way to phrase this in terms of the language above is that we are not modelling all of a specific feature (like scope) by mapping this to the λ-calculus without classes. On the other hand maybe this is a good thing?

4 Current Implementation

Here we detail the current Haskell implementation of the sketches above. We assume that a language is given as an Algebraic Data-type (ADT) with the recursive knot untied, namely, given an expression language like:

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

We expect it to be given as

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

Which is a straightforward transformation. Given this unrolled representation, it is possible to "tie the knot" manually with a `newtype`-ed Y-combinator, like so:

```
newtype Mu f = Mu (f (Mu f))

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

We use a simple extension to this, which to our knowledge is new, to allow for two ADTs to be interleaved:

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

We then use the popular `lens` library to execute the following workflow, given a language to model $\mathcal{L}_1$ and a language for which we have rules regarding referential transparency ($\mathcal{L}_0$).

(1) Write rules translating $\mathcal{L}_1$ variants into $\mathcal{L}_0$ ones. This need only have the type `a :+: b -> a :+: b`, so they should be shallow, and then `lens` can apply the rules until it reaches a fixed point. Inverse rules should also be written

(2) "Fuzz" using a QuickCheck Arbitrary instances, the subterms in $\mathcal{L}_0$ for referentially transparent ones (e.g. β-reduction/expansion)

(3) Apply the inverse rules, compare the output(s) to evaluating the unadulterated expression.

[9]While it is a common convention in python to start function names with a lower-case letter and class names with upper-case, this isn't true for, say, C-like languages where it depends on the style guide

Note that the inverse rules and mapping rules are very similar to the platonic ideal of just explaining what AST nodes are functions. This framework also allows for easily testing (using Quickcheck properties) whether or not the mappings are true inverses invariant under the substitutions made by step 2.

5 Orthogonality

A special case of the notion of sublanguage described above is the notion of feature interaction. It is considered desirable for a language feature to be implemented in an *orthogonal* manner, a term which is informally taken to mean "doesn't cause other features in the language to behave unexpectedly"[10]. We can model this has partitioning a given language into the language with and without the feature, and consider surprising cases in that context. While this may appear to be a mere exercise, it is very similar to the process of testing a new language implementation against preexisting programs.

The purpose of this section, however, is to show how it is possible to provide a language implementation that is orthogonal by construction; in fact, orthogonality will be a free theorem, assuming a program is written with the proper type signature. Consider an ADT for an untyped λ-calculus with simple types:

```
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
```

We can now annotate this ADT with feature-specific information, where `:++` is append for type-level lists.

```
-- Empty phantom types denoting present features
data UTLC
data Booleans
data Arithmetic
data Exp  :: [*] -> * where
  Id      :: String -> Exp [UTLC]
  Lam     :: String -> Exp a -> Exp (UTLC ': a)
  App     :: Exp a -> Exp b -> Exp (UTLC ': (a :++ b))
  ArithOp :: (Integer -> Integer -> Integer)
          -> Exp a -> Exp b -> Exp (Arithmetic ': (a :++ b))
  N       :: Integer -> Exp [Arithmetic]
```

[10]Another use-case is that of redundancy. Feature X is not orthogonal to feature Y if we can accomplish some class of programs with either X or Y. I don't think we cover that here

```
  BoolOp  :: (Bool -> Bool -> Bool)
          -> Exp a -> Exp b -> Exp (Booleans ': (a :++ b))
  IF      :: Exp a -> Exp b -> -> Exp c -> Exp (Booleans ': (a :++ b :++ c))
  B       :: Bool -> Exp [Booleans]
-- Existentially quantified expression type
data SomeExp = forall ls. SE (Exp ls)
data Val = NV Integer | BV Boolean | Clos SomeExp
```

We can then implement an interpreter feature-by-feature using a variant of the "extensible `letrec`" trick: Here is how we would implement the `Arithmetic` interpreter.

```
arithEval :: (forall a. Exp a -> Maybe Val) -> Exp (Arithmetic ': b) -> Maybe Val

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

utlcEval :: (forall a. Exp a -> Maybe Val) -> Exp (UTLC ': b)        -> Maybe Val
-- ...

boolEval :: (forall a. Exp a -> Maybe Val) -> Exp (Booleans ': b)    -> Maybe Val
-- ...

eval :: Exp a -> Maybe val
eval (Id s)           = utlcEval  eval (Id s)
eval (Lam s b)        = utlcEval  eval (Lam s b)
eval (App a b)        = utlcEval  eval (App a b)
eval (ArithOp f a b) = arithEval eval (ArithOp f a b)
eval (N n)            = arithEval eval (N n)
eval (BoolOp f a b)  = boolEval  eval (BoolOp f a b)
eval (IF c a b)       = boolEval  eval (If c a b)
eval (B b)            = boolEval  eval (B b)
```

Note now we can state a clean notion of sublanguage. A sublanguage is just an ADT with a type-level list containing some subset of the possible features. To prove orthogonality, we have to say that the evaluation of any sublanguage is compositional: it is identical regardless of what additional features are in the language. But this is just a free theorem given the type signature of the `*Eval` functions; they cannot possibly have an impact on additional features because they cannot inspect the abstract type parameter `b`! The remainder of the proof is to verify that `eval` does the obvious thing with each of these helper functions: a process that could easily be automated.

Hence, we have provided a scheme for building interpreters for languages that are orthogonal *by construction* and implement this without the need for full dependent types.

Doubts/Concerns There is something fishy about using parametricity when considering GADTs. Each of these helper functions can just pattern-match on all of the values of subterms. However, we could still get parametricity if we insist that they live in a separate module, and not require to be recompiled if `Exp` is augmented, but then we would require a stronger module system than what Haskell currently has. I think we are close to something useful though. Unrolling the data-type definition could help.

6 Denotational Setting

We can give the same definition in the context of denotational semantics. This gives a more concise characterization of surprise in terms of commutative diagrams.