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 √15748250656434120694561831064211130829461807819038434852 1
(sqrt 15748250656434120694561831064211130829461807819038434852 1)
; => 1254920342349829019399943221 1
```

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 √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(15748250656434120694561831064211130829461807819038434852 1)
# => 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 Practical Implementation

To make this definition practical, we can restrict ourselves to operating on the syntax of the language being modeled, provide syntactic mappings to a language *with* a semantics (e.g. the λ-calculus), and non-deterministically substitute equivalent terms in the second language into their counterparts in the first. Given an interpreter for the modeled language, we can then search for surprises in an automated fashion.

4 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.

[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