# Pixy Semantics

Reed Mullanix, Finn Hackett

February 2018

## 1 Introduction

The semantics of Pixy can be divided up into 3 portions: The term language, how that term language is evaluated, and the type system.

## 2 Term Language

The term language of Pixy is (roughly) as follows:

$\langle \mathit{expr} \rangle ::= \langle \mathit{literal} \rangle$
  $|$ $\langle \mathit{var} \rangle$
  $|$ `nil`
  $|$ `?` $\langle \mathit{expr} \rangle$
  $|$ `if` $\langle \mathit{expr} \rangle$ `then` $\langle \mathit{expr} \rangle$ `else` $\langle \mathit{expr} \rangle$
  $|$ $\langle \mathit{expr} \rangle$ `fby` $\langle \mathit{expr} \rangle$
  $|$ $\langle \mathit{expr} \rangle$ `where` $(\langle \mathit{var} \rangle$ `=` $\langle \mathit{expr} \rangle)*$
  $|$ `fun` $\langle \mathit{var} \rangle$ `=>` $\langle \mathit{expr} \rangle$
  $|$ $\langle \mathit{expr} \rangle$ $\langle \mathit{expr} \rangle$

NOTE: This is incomplete! We need to standardize on the term language.

## 3 Evaluation

The evaluation rules for Pixy are quite different from other languages. To begin with, each expression can be seen as a taking a State and producing a value and a new State. This state is then fed back into the expression to produce a new State and value, and so on. However, some expressions pose some problems. For example, when evaluating if ... then ... else expression, we should only really evaluate one of the branches, but doing so may skip important stateful evaluation inside of the untaken branch. To reconcile this, we present a model of evaluation which we call "Choked Evaluation". Whenever we are presented with a branching construct, we still evaluate the branches, with the caveat that all variables *and* literals evaluate to nil on the branch that is not taken.

Another point we need to make is that evaluation is only valid on **closed** expressions, or expressions that have no free variables.

NOTE: Insert full evaluation semantics here.

NOTE: We need to spec out when exactly evaluation terminates for a given step.

# 4 Type Theory

Typically, type systems follow this general form:

- The user declares the construction and elimination rules for a type.
- The user then uses theses construction rules to create programs.

We prefer to take a different approach, which has been strongly influenced by systems such as NuPRL. Generally speaking, our type system works as follows:

- The user writes a program.
- The user then creates a proof that the program inhabits some type.

That of course raises the question: When does a program inhabit a type? To answer that, we must first answer what exactly a type is in Pixy. We define a type as having 2 components:

1. A collection of canonical inhabitants.
2. An equivalence relation over those inhabitants.

For example, the canonical inhabitants of the type Nat are $0, 1, 2, 3...$ and the equivalence relationship is just the equivalence relationship of natural numbers. When we say that $a \in A$, what we are really saying is that $a = a$ under the equality relationship imposed by $A$. This point may seem slightly pedantic, but it has large implications. This can be extended to separate elements, so we could also propose that $a = b \in A$, or that 2 terms $a$ and $b$ are equivalent under the equality relation of $A$. Note that the canonical inhabitants aren't the only members of a type. Any term that evaluates to a canonical inhabitant is also a member of the type. On top of that, if we have 2 terms $t, t'$ and they evaluate to $a, a'$ respectively, and $a = a' \in A$, then $t, t' \in A$ as well!

Continuing in the spirit of NuPRL, what exactly is $a \in A$? Well, if we use the logic of Propositions-as-Types, $a \in A$ should really just be a type! We shall denote this type as $Eq\ a\ b\ A$. We shall also include all of the standard portions of Martin-Löf Type Theory.

NOTE: This section is incomplete, as we have multiple ways of preceding. I have listed out the possible options.

1. Use a temporally indexed dependent type. This allows us to encode certain properties such as "$\forall$ Times t, ..." and "$\exists$ Time t, ..." easily.
2. Use a co-inductive stream type. This would allow us to more easily prove relationships between 2 streams.

## 5 Relating Programs to Types

Note again that we do not derive the types of programs from the bottom-up, as is the norm. Rather, we prove that programs inhabit types from the top-down, using a proof refinement system.

To begin, a proof is a tree of **Judgments**, which consists of a number of **hypotheses** of the form $x : A$ followed by a **Goal**, which is of the form $term : T$. To proceed with the proof, we need to use refinement rules, which are ways of decomposing sub-goals. For example, say we had some term fun x => x, and we wanted to prove that this term is a member of $Bool \to Bool$. An example proof would be as follows:

```
H >> (fun x => x) in Bool -> Bool by intro-function.
    x:Bool, H >> x in Bool by hypothesis x.
    H >> Bool in U by bool-intro-universe.
```

Note that we use 3 rules here, $intro-function$, $hypothesis$, and $intro-universe$. These correspond to the standard type inference rules, but there is a catch: We cannot infer the types. This is because a term can inhabit many potential types. For example, we could also prove that fun x => x inhabits the type $\Pi_{A:U}.A \to A$:

```
H >> (fun x => x) in (A:U) -> A -> A by intro-function-pi
    A:U, x:A, H >> x in A by hypothesis x.
```

NOTE: The above rule needs some thinking about. As such, I have decided to not include it in the rule section yet.

NOTE: Write some examples that show how to use the rules to prove nil-safety.

## 6 Rules

### 6.1 Bool

```
H >> true in Bool by intro-true.
H >> false in Bool by intro-false
H >> Bool in U1 by bool-intro-universe.
```

### 6.2 Nil

```
H >> nil in Nil by intro-nil.
-- TODO: Write the choking type rules.
-- Needs clarification
```

### 6.3 Functions

```
H >> (fun x => b) in (x:A) -> B by intro-function.
```

```
    x:A, H >> b in B.
    H >> A in Ui.
    H >> B(x) in Ui. -- We may have to be careful about universe levels?
H >> (x:A) -> B in Ui by function-intro-universe.
    H >> A in Ui.
    H >> B in Ui.
```

### 6.4 Universes

```
H >> Ui in Uj by universe-cumulative.
    -- Note, i < j.
```