Pixy progress report

A. Finn Hackett, Reed Mullanix

April 25, 2018

Contents

1 Introduction

Our initial goal was to adapt the Lucid programming language to have first-class support for real-time programming, while retaining its nature as a general purpose language.

This means a few things: every step of computation must be able to meet a deadline, unrelated parts of a program must not be able to interfere with each others' timing and ideally even related parts of a program must never stop each other.

This requires a heavily asynchronous mindset for which Lucid is a good starting point, since almost everything in Lucid is indeed asynchronous. This also requires avoiding the original Lucid implementation's reliance on indexed streams, since they worked based on a heuristic [2, pages 67-68] that would induce arbitrary recomputation in some cases. [1]

Our plan was to derive our own set of evaluation semantics for the language that was to become Pixy that were able to compute all the values for each timestep using bounded memory and time, and then to work on enforcing more and more correctness properties such as co-inductive deadlock-freedom and custom invariants via a temporal type system.

Our progress so far is that we have designed Pixy' evaluation semantics from the ground up to operate using a fixed amount of state and compute every timestep within an arbitrary but constant deadline.

2 Pixy's programming model

We have defined Pixy's operational semantics in terms of nested state machines - any Pixy expression will evaluate to a state machine that will operate asynchronously, yielding a value at each timestep $t \in \mathbb{N}$. This approach allows for a much finer-grained control of the asynchronous relationships between dataflow nodes than Lucid's original lazy computation model, at the cost of some flexibility.

Specifically, Pixy does not support nested iteration in that same way that Lucid does, nor do custom operators enjoy the same degree of freedom as they did in Lucid.

To introduce Pixy's supported subset of Lucid, here is Pixy's grammar:

⟨*expr*⟩ ::= ⟨*number*⟩
| ⟨*var*⟩
| ⟨*bool*⟩
| `nil`
| `?` ⟨*expr*⟩
| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
| ⟨*expr*⟩ `fby` ⟨*expr*⟩
| ⟨*expr*⟩ `where {` ⟨*wheredecls*⟩ `}`
| `next` ⟨*expr*⟩
| ⟨*var*⟩ `(` ⟨*exprlist*⟩ `)`
| ⟨*expr*⟩ `+` ⟨*expr*⟩
| ⟨*expr*⟩ `-` ⟨*expr*⟩
| ⟨*expr*⟩ `*` ⟨*expr*⟩
| ⟨*expr*⟩ `/` ⟨*expr*⟩
| ⟨*exprtuple*⟩

[1] The original implementation would store the last 200 or so steps of the computation, recomputing anything older than that on demand. The heuristic was that this was unnecessary most of the time. This is hardly useful if we want precise bounds on the amount of memory a Pixy program might use or how long it will take.

⟨*exprtuple*⟩ ::= `(` ⟨*exprlist*⟩ (`|` ⟨*expr*⟩)? `)`

⟨*bool*⟩ ::= `true` | `false`

⟨*exprlist*⟩ ::= ⟨*expr*⟩ `,` ⟨*exprlist*⟩ | ⟨*expr*⟩

⟨*varlist*⟩ ::= ⟨*var*⟩ `,` ⟨*varlist*⟩ | ⟨*var*⟩

⟨*wheredecl*⟩ ::= ⟨*var*⟩ `=` ⟨*expr*⟩
| ⟨*var*⟩ `(` ⟨*varlist*⟩ `)` `=` ⟨*expr*⟩
| `(` ⟨*varlist*⟩ (`|` ⟨*var*⟩)? `)` `=` ⟨*expr*⟩

⟨*wheredecls*⟩ ::= ⟨*wheredecl*⟩ `;` ⟨*wheredecls*⟩ | ⟨*wheredecl*⟩

In principle, any Pixy expression with no free variables can be viewed as a self-contained program. Our current implementation reads a series of wheredecls and executes a main operator for the sake of ease of use.

You will notice that the grammar above contains ⟨*exprtuple*⟩, which can be used to construct tuples. This is a work in progress and will not be described in any depth in this paper, but it is expected to become part of the language in future work. Broadly speaking, tuples would be able to be constructed and manipulated as an extra data type. The only feature of note is that we expect that tuple literals will function as synchronisation points between streams during pre-delay calculations.

2.1 From an end-user point of view

When programming in Pixy, one can consider expressions as representing streams of values. This does not match our state-machine based formal model, but can aid in explaining its idiosyncrasies.

In this section we will discuss programs written in Pixy and how one can reason about them.

2.1.1 Literals

A single literal is the most trivial Pixy program. Any literal in Pixy will repeatedly yield itself. For example, 1 will yield the sequence $1, 1, 1, ...$ forever.

The same goes for *true* and $false$. The exception is *nil*, which describes an empty stream. It will never yield a value.

One can also bind Pixy expressions to names like so $x = 1$, in the context of which the expression x will also yield $1, 1, 1....$

2.1.2 Dataflow operators

Any Pixy expression is composed of sometimes-stateful dataflow operators. Many operators mirror common operations in mainstream programming languages like $+$, $-$, etc...

Pixy also features some more unique operators to control the flow of time.

fby We will first introduce fby: fby expresses sequencing. Combining fby with literals, one can write $1 \text{ fby } 2$. This program will yield the infinite sequence $1, 2, 2....$

Strictly speaking, fby will take one value from its left-hand side then yield all values from its right-hand side.

Mixing assignment with fby allows us to elegantly specify different recurrences:

```
x = 1 fby x+1
```

A quintessential example due to the ability to trivially identify which timestep any value of x came from, the expression x will yield the infinite increasing sequence $1, 2, 3, 4....$

To be precise, fby will yield one value from the expression 1, then all the values from the expression $x + 1$ which expresses the sequence $2, 3, 4....$

Likewise, it is quite easy to define alternating sequences using only fby:

```
x = 1 fby 2 fby 3 fby x
```

In this case the expression x will yield $1, 2, 3, 1, 2, 3....$

next The other important temporal operator in Pixy is the next operator. It allows us to express a lookahead into a stream.

For example, taking one of our previous examples $x = 1 \text{ fby } x+1$, the expression $\text{next } x$ will yield the infinite sequence $2, 3, 4....$

We can also use it in direct conjunction with fby to express more sophisticated sequences:

```
x = 1 fby 1 fby (x+ next x)
```

In this case, the expression x yields the sequence $1, 1, 2, 3, 5...$, or the Fibonacci sequence.

The intuition to how this works is to consider x to yield the "first element" of the stream and $\text{next } x$ to yield the "second element". So, initially both are 1, then as we move forward in time we have $1, 2$, and so forth. $x + \text{next } x$ simply calculates the sum of these two values going forward in time.

2.1.3 Where and user-defined operators

To help keep Pixy code organised and allow code reuse, we provide some primitives to control the scoping of values.

where Where clauses are similar to the construct of the same name and similar syntax in Haskell. They allow you to create a local scope with variables private to a specific expression.

Here is an example of a Pixy program using a where expression:

```
x = a + b
    where {
        a = 1 fby b + 1
        b = 2 fby a + 1
```

```
}
```

For more involved programs like this one, we encourage readers to check their behaviour using tables like these:

$a_1 = 1$	$b_1 = 2$	$x_1 = 3$
$a_2 = 3$	$b_2 = 2$	$x_2 = 5$
$a_3 = 3$	$b_3 = 4$	$x_3 = 7$
$a_4 = 5$	$b_4 = 4$	$x_4 = 9$
...		

This table for example shows that we have defined a Rube Goldberg-esque odd number generator.

This also exercises the key features of where expressions: aside from allowing assigning expressions to variables and then using them in the $a+b$, where expressions allow defining complex mutually-referential streams.

It is also worth noting that definition order has no impact on program semantics - the following program is semantically identical to the first:

```
x = a + b
    where {
        b = 2 fby a + 1
        a = 1 fby b + 1
    }
```

User-defined operators As well as giving names to streams, Pixy also allows defining custom operators. Syntactically they are similar to function definition and application, but they have an important difference: they operate on streams, so we call them operators.

Here is an example program equivalent to the example above that demonstrates custom operator definition:

```
f1(p, q) = p fby q

x = a + b
    where {
        f2(a, b) = a fby b + 1
        a = f1(1, b+1)
        b = f2(2, a)
    }
```

Custom operators can be visualised as operating by substitution. They can be defined at any location and do not perform any kind of variable capture - their inner scope is composed entirely of their arguments. $f2$ demonstrates that you can safely use the same

name to refer to a parameter and a variable defined outside an operator and not run into conflicts.

Custom operators do not support recursion at the moment - that is a topic for future work.

2.1.4 Conditionals

Pixy also features an if expression syntactically similar to those found in Haskell or ML. Comparison past superficial syntax may be misleading however.

Often, Pixy's conditionals act like one might naively expect:

```
y = 1 fby y + 1
x = if y % 2 == 0 then y else 0
```

This will unsurprisingly cause the expression x to yield the sequence $0, 2, 0, 4, 0, 6....$

The intuition that the condition is evaluated, then based on the result one of the two branches is returned is largely correct.

Now consider this code:

```
y = 1 fby y + 1
x = if y % 2 == 0
    then
        z where { z = 1 fby z + 1 }
    else
        0
```

This yields the same sequence, but highlights an important detail: time progresses as the same rate in each branch of the if statement, regardless of which branch is chosen. In effect, an if expression acts as a 2-way multiplexer.

Choking and nil Now let us introduce *nil* more fully and demonstrate a confusing but important extra detail of if expressions.

Consider the following program:

```
y = 1 fby y + 1
x = if y % 2 == 0 then y else nil

z = if ?x then x + 1 else 0
```

In this program, z will yield the sequence $3, 5, 7....$ This may raise a few questions.

First, we need to explain that $?x$ is a new operator. It is called a nil-check and maps from a stream of values to a stream of booleans. If a given value in the original stream was *nil* then that element of the new stream will be $false$, otherwise the stream will be comprised of repeated *true*.

The nil-check is useful for reacting to a stream not yielding a value at the current timestep. Formally, x does not yield a value when y yields a value that is odd. When a stream does not yield a value its value is called *nil*, and the only thing you can do to

that value is check for it and forward it. In the case of z, we chose to default z's value to 0 when x does not have a value.

This then raises the second question: if an if expression evaluates both of its arguments, why wouldn't the program crash on evaluating $nil + 1$, even if the 0 is chosen at that time?

The answer is the concept of choke-evaluation. If if expressions really did evaluate both their arguments then the problem would have no solution - you would not be able to shield a given subexpression from encountering *nil* values.

Choke evaluation is applied to a non-taken branch of an if expression in order to make sure any state it has progresses, while "pretending" to not evaluate it. That is, a choke-evaluated expression must always yield *nil* but may perform upkeep internally if need be. We achieve this by defining a second mode of evaluation where all literals and variable references return *nil*, as well as defining semantics for all the operators to tolerate *nil* under most circumstances.

For example, in the above program what is really evaluated when x is *nil* is $nil + nil$, which evaluates to *nil* without any issue.

The closest thing to an "exception" to choke-evaluation is when you try to choke-evaluate a where expression. As before, it is required that the expression yields *nil* in the end, but all the internal definitions are still evaluated.

This is why the following code from the previous section also does what one would expect - z is still updated, since the expression that updates it exists inside a where expression:

```
y = 1 fby y + 1
x = if y % 2 == 0
    then
        z where { z = 1 fby z + 1 }
    else
        0
```

We find that choke-evaluation strikes a balance between the ability to skip evaluation of a non-taken branch and the ability to keep time synchronised between both branches of an if expression, despite its sometimes strange-looking behaviour.

2.2 Operational semantics

Pixy's operational semantics have several stages in order to keep memory allocation and analysis distinct from the actual evaluation steps. This helps keep evaluation simple and allows us to prove quite trivially that for every timestep a Pixy program will perform a bounded amount of computation. We chose to avoid time-indexed computation in order to stay in keeping with the idea that a Pixy program only needs its immediate state in order to keep functioning - that is to say, a Pixy program could in theory be coinductively verified to remain correct and on time by symbolically evaluating reachable states, and will run using a well-defined finite amount of memory.

The first stage is used to flatten out any operator calls. Since recursive operator calls could lead to unbounded memory usage during evaluation, Pixy requires all operator calls including recursive ones to be flattened out before evaluation. In order to predict the maximum possible recursion depth and allocate memory for it ahead of time, we use a technique based on sized types.

The second stage is the allocation of initial state for all builtin operators. This stage has two subsections, namely state allocation and delay compensation.

The last stage is the evaluation derivations themselves, which operate on the modified AST and initial state defined in the prior stages. This stage models the life cycle of the program using what can be considered both small- and big-step semantics depending on your point of view. If you consider the program's entire lifespan, then we express that in a series of small steps. If you consider one step, we provide big-step semantics for analysing one single step. To avoid confusion, we will notate evaluation using $\rightarrow$.

2.2.1 Operator calls

To begin with, we flatten out all operator calls via substitution. This is both to ensure that there is a clear bound on the time a Pixy expression can take to perform one time step and to simplify the formulation of any further stages.

We have an idea of how to deal with recursive operator application, but it is incomplete. Similar to [1], we expect to reason about recursion depth using sized types. Specifically, we hope to generate upper and lower bounds for all operator arguments and enforce that the range of possible inputs to a recursive operator strictly decreases as recursion depth increases. Thus, we should be able to either generate a series of substitutions equivalent to the deepest recursion depth possible (terminating in an infinitely choked if branch) or reject an expression as ill-formed.

Our problem at the moment is that we do not have a precise enough representation of numbers in Pixy. While it is simple enough to calculate upper and lower bounds for an expression given its type, performing operations on these bounds requires a precise definition of numerical edge cases that we lack. We leave to future work exactly how to do this, but expect that once a more precise set of numerical semantics are derived the issue will be simpler.

For the simple case where we assume no recursion occurs, operator call flattening can be expressed using the syntax $\Gamma \vdash E \overset{\text{flatten}}{\Rightarrow} E'$ where Γ represents the current scope, E represents the initial expression and E' represents the modified result with no operator calls.

$$\frac{\begin{array}{c}\Gamma \vdash B \overset{\text{flatten}}{\Rightarrow}_{W1} B'; \Gamma' \\ \Gamma, \Gamma' \vdash B' \overset{\text{flatten}}{\Rightarrow}_{W2} B'' \\ \Gamma, \Gamma' \vdash E \overset{\text{flatten}}{\Rightarrow} E'\end{array}}{\Gamma \vdash E \,\text{where}\{B\} \overset{\text{flatten}}{\Rightarrow} E' \,\text{where}\, B''}[\text{Flatten-where}]$$

Since order inside a where clause is arbitrary, we have to be careful how we treat scoping. First we pass through and collect all the operator definitions, then we perform a second pass while applying the operators wherever necessary.

$$\frac{\Gamma \vdash R \overset{\text{flatten}}{\Rightarrow}_{W1} R'; \Gamma'}{\Gamma \vdash F(A...) = E; R \overset{\text{flatten}}{\Rightarrow}_{W1} R'; F(A...) = E, \Gamma'} [\text{Flatten-where-1-operator}]$$

$$\frac{\Gamma \vdash R \overset{\text{flatten}}{\Rightarrow}_{W1} R'; \Gamma'}{\Gamma \vdash V = E; R \overset{\text{flatten}}{\Rightarrow}_{W1} (V = E; R'); \Gamma'} [\text{Flatten-where-1-variable}]$$

$$\frac{}{\Gamma \vdash \overset{\text{flatten}}{\Rightarrow}_{W1} \emptyset} [\text{Flatten-where-1-empty}]$$

Once we've collected all the operator definitions, we traverse all the subexpressions and substitute them in when necessary:

$$\frac{\begin{array}{c}\Gamma \vdash E \overset{\text{flatten}}{\Rightarrow} E' \\ \Gamma \vdash R \overset{\text{flatten}}{\Rightarrow}_{W2} R'\end{array}}{\Gamma \vdash V = E; R \overset{\text{flatten}}{\Rightarrow}_{W2} V = E'; R'} [\text{Flatten-where-2-variable}]$$

$$\frac{}{\Gamma \vdash \overset{\text{flatten}}{\Rightarrow}_{W2}} [\text{Flatten-where-2-empty}]$$

When we encounter an operator application, we replace it the operator's body and a scope modifier to avoid name conflicts:

$$\frac{\begin{array}{c}\Gamma(F(A...) = E) \\ \Gamma \vdash E \overset{\text{flatten}}{\Rightarrow} E'\end{array}}{\Gamma \vdash F(V...) \overset{\text{flatten}}{\Rightarrow} [A = V...]E'} [\text{Flatten-apply}]$$

Note: the scope modifier pictured here is not a substitution - it will be interpreted specially by later stages.

2.2.2 Delay compensation

Often, delay compensation must be applied to the evaluation of a Pixy expression in order for it to operate correctly.

To give a brief rationale, consider the following Pixy snippet:

```
x = 1
y = x + next x
```

Following a naive reading of the evaluation semantics, at t_1 x will be 1, and when computing y we will take 1 and try to add it to next x, which dropped the value 1 and is equal to *nil* in this case. This is counter-intuitive since $x +$ next x looks like it means "wait for next x" then add x to it. When considering this issue we found that it was possible to reimplement the intuitive interpretation using implicit buffering, or delay compensation.

Initially we attempted to express this in terms of ad-hoc buffering at the offending operator, + in this case. This however encountered problems when under choke-evaluation. Take the following expression:

```
if ?x then (x+x) + next x else 1
```

Assuming x may be *nil*, the problem here is that the if statement will try to choke-evaluate $(x + x)$ on the wrong timestep. While the if expression should be preventing $(x + x)$ from being computed when x is nil, the value of x would reach the condition clause one timestep after $(nil + nil)$ had been evaluated and the Pixy program had crashed. This is because making + buffer its inputs assumes that the operands will be valid at every timestep, which may not be true.

To fix this issue, we ensure that only variable references are buffered. No computation occurs when referencing a variable, so there can be no errors due to incorrectly timed *nil* values.

To achieve this, we specify a delay compensation relation that may be used by the init stage in order to add buffers to nested variable references.

The syntax for the relation looks like this: $\mathcal{V}; d; \delta; S \vdash E \overset{\text{compensate}}{\Rightarrow} S'; \delta'_e; \Delta'_e$.

Unspecified letters have the same function as those used in the init stage.

S refers to the existing initial state computed for expression E.

δ refers to the amount by which any identifiers are to be delayed on top of their existing delay.

S' refers to the new initial state after delay compensation has been computed.

δ'_e refers to the new delay of the expression E after compensation.

Δ'_e refers to the constraints applicable to the delays in E after compensation.

Id The only derivation we will give for compensation is that for identifiers - all others are identical to the init stage except for passing along the extra information present during delay compensation.

What happens for identifiers is that the extra delay δ is applied to the identifier's buffer. As you will see from the init stage, this starts as buffer of length 0 and becomes longer as needed to buffer enough of a variable's history.

In principle one could hold a single common buffer for every identifier and grow/index into it as needed, but this would significantly complicate the topics so we will leave that as an exercise for the implementation.

$$\frac{I \in \mathcal{V}}{\mathcal{V}; d; \delta; \mathcal{B}[N] \vdash I \overset{\text{compensate}}{\Rightarrow} \mathcal{B}[N+\delta]; \delta(I)+1; \emptyset}\text{[Compensate-id-where]}$$

$$\frac{I \notin \mathcal{V}}{\mathcal{V}; d; \delta; \mathcal{B}[N] \vdash I \overset{\text{compensate}}{\Rightarrow} \mathcal{B}[N+\delta]; \delta(I); \emptyset}\text{[Compensate-id-default]}$$

2.2.3 init stage

Pixy has an initialisation stage during which all the state needed for any later computation will be initialised and buffers are put in place to implement delay compensation - this is the only stage that controls the allocation of memory.

The syntax for derivations of this stage looks like this:

$$\mathcal{V}; d \vdash E \overset{\text{init}}{\Rightarrow} S; \delta; \Delta$$

$\mathcal{V}$ is the set of names that are currently being defined by a where expression.

d is the number of nested fby expressions containing E (initially 0)

S is the starting state for expression E

δ is the pre-delay of expression E, that is, the number of timesteps before E will yield values.

Δ is the set of constraints on pre-delay time. These are to be solved by a constraint solver in order to derive the appropriate pre-delays of any expression.

Binop Binary operators are one of the main places where delay compensation occurs. Their state allocation is simple - we recursively allocate enough space for the operands. The important detail however is that we must ensure the operands are synchronised. If their δ values are different, we must apply delay compensation.

$$\frac{\begin{array}{c}\mathcal{V}; d \vdash A \overset{\text{init}}{\Rightarrow} S_a; \delta_a; \Delta_a \\ \mathcal{V}; d \vdash B \overset{\text{init}}{\Rightarrow} S_b; \delta_b; \Delta_b \\ \mathcal{V}; d; \max(\delta_a, \delta_a) - \delta_a; S_a \vdash A \overset{\text{compensate}}{\Rightarrow} S'_a; \delta'_a; \Delta'_a \\ \mathcal{V}; d; \max(\delta_a, \delta_a) - \delta_b; S_b \vdash B \overset{\text{compensate}}{\Rightarrow} S'_b; \delta'_b; \Delta'_b\end{array}}{\mathcal{V}; d \vdash A \; op \; B \overset{\text{init}}{\Rightarrow} (S'_a, S'_b); \max(\delta'_a, \delta'_b); \Delta'_a, \Delta'_b}\text{[Init-binop]}$$

In each case we compensate the faster of the two operands to be slower - this ensures that only values from the same time will meet each other.

Conditionals If statements also require delay compensation, this time across 3 different operands. Otherwise, we just recursively allocate space for the conditional and two branches.

$$\frac{\begin{array}{c}\mathcal{V}; d \vdash C \overset{\text{init}}{\Rightarrow} S_C; \delta_C; \Delta_C \\ \mathcal{V}; d \vdash T \overset{\text{init}}{\Rightarrow} S_T; \delta_T; \Delta_T \\ \mathcal{V}; d \vdash F \overset{\text{init}}{\Rightarrow} S_F; \delta_F; \Delta_F \\ \mathcal{V}; d; m - \delta_C; S_C \vdash C \overset{\text{compensate}}{\Rightarrow} S'_C; \delta'_C; \Delta'_C \\ \mathcal{V}; d; m - \delta_T; S_T \vdash T \overset{\text{compensate}}{\Rightarrow} S'_T; \delta'_T; \Delta'_T \\ \mathcal{V}; d; m - \delta_F; S_F \vdash F \overset{\text{compensate}}{\Rightarrow} S'_F; \delta'_F; \Delta'_F\end{array}}{\mathcal{V}; d \vdash \text{if } C \text{ then } T \text{ else } F \overset{\text{init}}{\Rightarrow} (S'_C, S'_T, S'_F); m'; \Delta'_C, \Delta'_T, \Delta'_F}\text{[Init-if]}$$

where $m = \max(\delta_C, \delta_T, \delta_F)$ and $m' = \max(\delta'_C, \delta'_T, \delta'_F)$

Next Since next needs to remember how many values it needs to skip in order to reach the second value yielded by E, we store that number. The number in question happens to be $\delta + 1$, since it is one more than the time needed to wait for the first value yielded by E. We also store the initial state of its only operand E.

$$\frac{\mathcal{V}; d \vdash E \overset{\text{init}}{\Rightarrow} S; \delta; \Delta}{\mathcal{V}; d \vdash \text{next } E \overset{\text{init}}{\Rightarrow} (\delta + 1, S); \delta + 1; \Delta}\text{[Init-next]}$$

Fby Fby is the only binary operator in Pixy that does not perform typical delay compensation. Since it is used for sequencing, instead we ensure that exactly one value from A can be sequenced before all the values from B.

We do this with a delay-based buffer but use a different strategy. We allocate a buffer internal to the fby expression's state that holds $\max(0, \delta_a - \delta_b + 1)$ elements. This means that if the delays for A and B are equal, that is, they are expected to arrive at the same time, then B's stream should be delayed by one timestep. If A is at all earlier than B, then B does not have to wait at all and the buffer size becomes 0 (pass-through). If A is later than B, B should wait one more than the number of timesteps A is late by, since it needs to wait for both A to arrive and be yielded (which takes one extra timestep).

We also make the first element of the state $false$ in order to initialise the latch between reading from A and reding from B. See the evaluation rules for how this is used.

$$\frac{\begin{array}{c}\mathcal{V}; d \vdash A \overset{\text{init}}{\Rightarrow} S_a; \delta_a; \Delta_a \\ \mathcal{V}; d + 1 \vdash B \overset{\text{init}}{\Rightarrow} S_b; \delta_b; \Delta_b\end{array}}{\mathcal{V}; d \vdash A \text{ fby } B \overset{\text{init}}{\Rightarrow} (false, \mathcal{B}[\max(0, \delta_a - \delta_b + 1)], S_a, S_b); \delta_a; \Delta_a, \Delta_b, \delta_a \geq \delta_b - d}\text{[Init-fby]}$$

To explain the constraint $\delta_a \geq \delta_b - d$ that we add to the delay constraint set Δ_a, Δ_b, this is to prevent $x = 1 \text{ fby next } x$ from being valid. In effect, that Pixy definition defines a stream whose first element is 1 and whose second element is ... its own first element, which has no computable value. When viewed from angle of delay constraints, such a situation would lead to $\delta_a = 0, \delta_b = 1$ and the constraint $0 \geq 1$ which fails. The other case that would fail this constraint is when B is late - this would insert a *nil* value into the output of the fby between the first value of A and the first value of B. This would be unintuitive implicit behaviour, so we forbid it in the current version of the language.

To explain the $-d$ term, it is to deal with constraints on nested fby expressions. Given the definition $x = 1 \text{ fby } 2 \text{ fby next } x$, solving the constraint without $-d$ would yield $\delta_a = 0, \delta_b = 1$ and the unsatisfied constraint $0 \geq 1$. This is undesirable, since this definition does work and defines the sequence $1, 2, 2, 2....$ This is fixed by changing the constraint to $0 \geq 1 - 1$. The intuition behind subtracting d, the number of nested fby expressions we are in, is that we know there is an enclosing fby that will hide at most d nils from the user, so B being late by at most d timesteps is permissible in this case.

Id When an identifier is encountered, we need to initialise the variable's buffer in case delay compensation needs to be done. Since it is possible that no compensation at all is required, we initialise the buffer to 0. In the delay compensation derivation for identifiers, this is why initially $N = 0$.

$$\frac{I \in \mathcal{V}}{\mathcal{V}; d \vdash I \overset{\text{init}}{\Rightarrow} \mathcal{B}[0]; \delta(I) + 1; \emptyset} \text{[Init-id-whereclause]}$$

$$\frac{I \notin \mathcal{V}}{\mathcal{V}; d \vdash I \overset{\text{init}}{\Rightarrow} \mathcal{B}[0]; \delta(I); \emptyset} \text{[Init-id-nodelay]}$$

The reason there are two versions of this derivation is due to pre-delay: if we are still among the where clauses where the variable was defined, we will access it with a delay of 1 since all where clauses run synchronously and all variables defined by a where clause will be *nil* at time 0. Otherwise, such as when a variable is passed as an argument to an operator or when we are in the body of a where expression, the value will be available immediately so we do not introduce any additional delays.

Operator scope Initialising operator scope is deceptively simple. The main issue we need to watch out for is that operator application acts as a synchronisation point - in order to ensure we pass in sets of values from the same point in time we must perform delay compensation on the operands.

$$\frac{\dfrac{\dfrac{\mathcal{V} \setminus \{A...\}; d \vdash E \overset{\text{init}}{\Rightarrow} S; \delta; \Delta}{\mathcal{V}; d \vdash V \overset{\text{init}}{\Rightarrow} S_v; \delta_v; \Delta_v}\,V...}{\mathcal{V}; d; \max(\delta_v...) - \delta_v; S_v \vdash V \overset{\text{compensate}}{\Rightarrow} S'_v; \delta'_v; \Delta_v}\,V...}{\mathcal{V}; d \vdash [A = V...]E \overset{\text{init}}{\Rightarrow} (S, (S'_v...)); \delta; \Delta, \Delta'_v..., \delta(A) = \delta'_v...}\,\text{[Init-operator-scope]}$$

Where During init, where expressions must enforce scoping rules and allocate the state of all their subexpressions.

We only add the names $\{N...\}$ to $\mathcal{V}$ when recursing over the assignments, since where expressions imply an ordering between their definitions and their body. The only effect of changing $\mathcal{V}$ is that identifiers in the set will gain $+1$ to their delay as references between different names being defined in a where expression are effectively buffered by the initial *nil* value they are set to during init. This helps keep order-independent evaluation of where expressions simple, since all identifiers refer to the previous value of that variable instead of the currently computed one.

$$\frac{\dfrac{\dfrac{}{\mathcal{V} \cup \{N...\}; d \vdash V \overset{\text{init}}{\Rightarrow} S_v; \delta_v; \Delta_v}\,V... \qquad \mathcal{V}; d \vdash E \overset{\text{init}}{\Rightarrow} S; \delta; \Delta}{\mathcal{V}; d \vdash E \,\text{where}\{N = V...\} \overset{\text{init}}{\Rightarrow} (S, (nil...), (S_v...)); \delta; \Delta, \Delta_v..., \delta(N) = \delta_v...}\,\text{[Init-where]}$$

To give an example of the effect of $\mathcal{V}$ in practice, consider the following set of mutually referential definitions:

```
x = 1 fby x + 1
y = x
z = y fby x
```

Due the way mutual references (even non-recursive ones) work, seeing an execution trace of each variable value at each timestep may be initially confusing:

$x_0 = nil$	$y_0 = nil$	$z_0 = nil$
$x_1 = 1$	$y_1 = nil$	$z_1 = nil$
$x_2 = 2$	$y_2 = 1$	$z_2 = nil$
$x_3 = 3$	$y_3 = 2$	$z_3 = 1$
$x_4 = 4$	$y_4 = 3$	$z_4 = 1$
$x_5 = 5$	$y_5 = 4$	$z_5 = 2$
...		

All the values seem to be out of step with each other.

Once you take into account delay compensation however, we realize that within this scope $\delta_x = 1, \delta_y = 2, \delta_z = 3$. This explains the apparent asynchrony: each expression is being evaluated one timestep later than the other, and buffering is used in various places to keep each expression from seeing any values from the wrong timestep.

A further exercise is to then consider the body of this hypothetical where expression, which is initialised without $\{x, y, z\}$ added to $\mathcal{V}$:

```
z where
    x = 1 fby x + 1
    y = x
    z = y fby x
```

Now, if we name the output of this expression p we can see how timings in the body of a where expression relate to those in the definitions:

$x_0 = nil$	$y_0 = nil$	$z_0 = nil$	$p_0 = nil$
$x_1 = 1$	$y_1 = nil$	$z_1 = nil$	$p_1 = nil$
$x_2 = 2$	$y_2 = 1$	$z_2 = nil$	$p_2 = 1$
$x_3 = 3$	$y_3 = 2$	$z_3 = 1$	$p_3 = 1$
$x_4 = 4$	$y_4 = 3$	$z_4 = 1$	$p_4 = 2$
$x_5 = 5$	$y_5 = 4$	$z_5 = 2$	$p_5 = 3$
...			

How come p appears to be predicting the future? A partial hint is in the delay for p: $\delta_p = 2$. Since the body of a where expression is executed strictly after all the definitions, the body of the where expression will see all the values of the definitions in the same timestep that they are computed, not one timestep later as with mutual references between where definitions. In fact, in the body the apparent delay of x, y and z is one shorter: $\delta_x = 0, \delta_y = 1, \delta_z = 2$.

2.2.4 Runtime evaluation semantics

Now that we have described how to prepare a Pixy program to be run, here are the semantics of a running Pixy program.

Since a Pixy program's evaluation takes place over an infinite series of timesteps, the evaluation semantics describe a single timestep in detail.

The syntax of one evaluation step looks like this: $\Gamma; S \vdash E \to V; S'$. Choke-evaluation has the same syntax, except that the relation is changed from $\to$ to $\overset{\text{C}}{\to}$.

Γ represents a mapping of variable names to value, that is, the current scope.

S represents the accumulated state associated with the current expression. It is important to note that the structure of state is strongly enforced: $(1, 2, 3)$ and $(1, (2, 3))$ are semantically different states, one being a tuple of three integers and the other being a 2-tuple of 1 and $(2, 3)$. When we pattern-match on state we rely on this property in order to avoid ambiguities and keep the states of different subexpression strictly separate.

E represents the current expression.
V is the value that E evaluates to in the context of Γ and S at the current timestep.
S' is the state that will be used to evaluate E in the next timestep.

Conditionals These evaluations are a key part of the formal description of choke-evaluation. If expressions are one of the few expressions whose non-choke evaluation rules initiate choke-evaluation.

Specifically, if expressions choke the non-taken branch. Otherwise, they act just like a ternary operator, advancing the state of their subexpressions and correctly multiplexing values from T and F.

$$\frac{\begin{array}{c}\Gamma; S_c \vdash C \to true; S'_c \\ \Gamma; S_t \vdash T \to V; S'_t \\ \Gamma; S_f \vdash F \xrightarrow{\text{C}} nil; S'_f\end{array}}{\Gamma; (S_c, S_t, S_f) \vdash \text{if } C \text{ then } T \text{ else } F \to V; (S'_c, S'_t, S'_f)}\text{[Eval-if-true]}$$

$$\frac{\begin{array}{c}\Gamma; S_c \vdash C \to true; S'_c \\ \Gamma; S_t \vdash T \xrightarrow{\text{C}} nil; S'_t \\ \Gamma; S_f \vdash F \to V; S'_f\end{array}}{\Gamma; (S_c, S_t, S_f) \vdash \text{if } C \text{ then } T \text{ else } F \to V; (S'_c, S'_t, S'_f)}\text{[Eval-if-false]}$$

$$\frac{\begin{array}{c}\Gamma; S_c \vdash C \to nil; S'_c \\ \Gamma; S_t \vdash T \xrightarrow{\text{C}} nil; S'_t \\ \Gamma; S_f \vdash F \xrightarrow{\text{C}} nil; S'_f\end{array}}{\Gamma; (S_c, S_t, S_f) \vdash \text{if } C \text{ then } T \text{ else } F \to nil; (S'_c, S'_t, S'_f)}\text{[Eval-if-nil]}$$

To cover all cases, we also specify that if an if statement's conditional receives *nil* then it chokes both its branches and yields *nil*. This is to ensure that expressions are *nil*-safe within reason. In other words, to ensure that expressions "wait" when not provided with input values by yielding *nil* themselves.

$$\frac{\begin{array}{c}\Gamma; S_c \vdash C \xrightarrow{\text{C}} nil; S'_c \\ \Gamma; S_t \vdash T \xrightarrow{\text{C}} nil; S'_t \\ \Gamma; S_f \vdash F \xrightarrow{\text{C}} nil; S'_f\end{array}}{\Gamma; (S_c, S_t, S_f) \vdash \text{if } C \text{ then } T \text{ else } F \xrightarrow{\text{C}} nil; (S'_c, S'_t, S'_f)}\text{[Eval-if-C]}$$

Binop This is a catch-all derivation for any binary operator in Pixy. Binary operators recurse over their arguments and perform their respective operation. They pass through choke evaluation, and have no effect when passed 2 *nil* values, returning *nil* themselves.

$$\frac{\begin{array}{c}\Gamma; S_l \vdash L \to V_l; S'_l \\ \Gamma; S_r \vdash R \to V_r; S'_r\end{array}}{\Gamma; (S_l, S_r) \vdash L\ op\ R \to V_l\ op\ V_r; (S'_l, S'_r)}\text{[Eval-binop]}$$

$$\frac{\begin{array}{c}\Gamma; S_l \vdash L \xrightarrow{\text{C}} nil; S'_l \\ \Gamma; S_r \vdash R \xrightarrow{\text{C}} nil; S'_r\end{array}}{\Gamma; (S_l, S_r) \vdash L\ op\ R \xrightarrow{\text{C}} nil; (S'_l, S'_r)}\text{[Eval-binop-C]}$$

Where Where expressions perform 3 functions:

1. Reading their scope from their saved state as part of our system to implement mutual reference between definitions in where expressions
2. Recursively evaluating all their definitions, then their body (in that order)
3. When choked, they only choke their body. The definitions are still evaluated as normal in order to avoid the state inside the where lagging behind the rest of the program.

$$\frac{\begin{array}{c}\overline{\Gamma, \overline{N = S_v...}^{S_v...}; S_s \vdash E_v \to V_v; S'_s}^{E_v...} \\ \Gamma, N = V_v...; S \vdash E \to V; S'\end{array}}{\Gamma; (S, (S_v...), (S_s...)) \vdash E\,\text{where}\{N = E_v...\} \to V; (S', (V_v...), (S'_s...))}\text{[Eval-where]}$$

$$\frac{\begin{array}{c}\overline{\Gamma, \overline{N = S_v...}^{S_v...}; S_s \vdash E_v \to V_v; S'_s}^{E_v...} \\ \Gamma, N = V_v...; S \vdash E \xrightarrow{\text{C}} nil; S'\end{array}}{\Gamma; (S, (S_v...), (S_s...)) \vdash E\,\text{where}\{N = E_v...\} \xrightarrow{\text{C}} nil; (S', (V_v...), (S'_s...))}\text{[Eval-where-C]}$$

Operator scope Operator scope has similar properties to a where expression, only they do not store any state. They simply ensure that all the arguments to the operator have been evaluated then add the results to the body's scope, evaluating it in turn.

$$\frac{\begin{array}{c}\overline{\Gamma; S_v \vdash E_v \to V_v; S'_v}^{E_v...} \\ A = D_v...; S \vdash E \to V; S'\end{array}}{\Gamma; (S, (S_v...)) \vdash [A = E_v...]E \to V; (S', (S'_v...))}\text{[Eval-apply]}$$

Similar to a where expression, operator scopes negate choking. Opposite to where expressions however, it is the arguments that are choked whereas the body is evaluated normally. It is up to the implementer of the operator to ensure that the operator does not naively use its arguments when they might be nil.

$$\frac{\begin{array}{c}\overline{\qquad\qquad}^{E_v...}\\ \Gamma; S_v \vdash E_v \xrightarrow{\text{C}} nil; S'_v \\ A = nil...; S \vdash E \to V; S'\end{array}}{\Gamma; (S, (S_v...)) \vdash [A = E_v...]E \xrightarrow{\text{C}} nil; (S', (S'_v...))}\text{[Eval-apply-C]}$$

fby Evaluating fby has some extra requirements compared to other operators. One one hand, it operates as a latch waiting for one value from A then choking A and exclusively forwarding values from B. On the other hand, it buffers values from B while waiting for the first value from A as well as when dealing with its backlog of values from B after it is done waiting for A.

Here we use **buffer** to represent cycling the fby's buffer b. The semantics of this operation are the same as when buffering identifier values - the buffer is assumed to start full of nil values, and the buffer acts like a queue. As you may notice in fby-A, the expectation is that the buffer is large enough to not run out of nil values before switching to fby-B. It is an error in the init stage for this not to be the case.

$$\frac{\begin{array}{c}\Gamma; S_a \vdash A \to V_a; S'_a \\ \Gamma; S_b \vdash B \to V_b; S'_b \\ \textbf{buffer}(b, V_b) = b', nil\end{array}}{\Gamma; (false, b, S_a, S_b) \vdash A \,\text{fby}\, B \to V_a; (V_a \neq nil, b', S'_a, S'_b)}\text{[Eval-fby-A]}$$

In fby-A you will notice that the first element of that state is pattern-matched to be $false$. This means the fby expression is still waiting for the first value from A. Correspondingly, the new value for that part of the state is whether a value was received from A, or $V_a \neq nil$. When that becomes true, that part of the state will be latched to $true$ and future evaluations will use fby-B instead.

In fby-B, we continue cycling the buffer with new values from B. The difference here is that the buffer functions purely as a delay with constant size - the deadline for receiving a value from A was the only thing requiring it to be a certain size so now we just forward whatever comes out of the buffer.

$$\frac{\begin{array}{c}\Gamma; S_a \vdash A \xrightarrow{\text{C}} nil; S'_a \\ \Gamma; S_b \vdash B \to V_b; S'_b \\ \textbf{buffer}(b, V_b) = b', V\end{array}}{\Gamma; (true, b, S_a, S_b) \vdash A \,\text{fby}\, B \to V; (true, b', S'_a, S'_b)}\text{[Eval-fby-B]}$$

When choked, fby chokes its arguments and does not update anything. This can happen irrespective of its latch state or buffer contents, since we forced A and B to yield nil and all of fby's transitions depend on non-nil values from one of the operands.

$$\frac{\begin{array}{c}\Gamma; S_a \vdash A \xrightarrow{\mathrm{C}} nil; S'_a \\ \Gamma; S_b \vdash B \xrightarrow{\mathrm{C}} nil; S'_b\end{array}}{\Gamma; (C, b, S_a, S_b) \vdash A \,\mathrm{fby}\, B \xrightarrow{\mathrm{C}} nil; (C, b, S'_a, S'_b)}\text{[Eval-fby-C]}$$

next Next expressions need to ensure that they drop the first non-nil value they receive. To do this, they choke E while counting down its pre-delay (N, stored in the state by the init stage) plus one, thus beginning to yield value from E on the timestep after E has yielded its first value.

$$\frac{\begin{array}{c}\Gamma; S \vdash E \xrightarrow{\mathrm{C}} nil; S' \\ N > 0\end{array}}{\Gamma; (N, S) \vdash \mathrm{next}\, E \to nil; (N-1, S')}\text{[Eval-next-countdown]}$$

$$\frac{\Gamma; S \vdash E \to V; S'}{\Gamma; (0, S) \vdash \mathrm{next}\, E \to V; (0, S')}\text{[Eval-next-forward]}$$

When choked, next still counts down. This is to ensure the timing matches the computed pre-delay.

$$\frac{\begin{array}{c}\Gamma; S \vdash E \xrightarrow{\mathrm{C}} nil; S' \\ N > 0\end{array}}{\Gamma; (N, S) \vdash \mathrm{next}\, E \xrightarrow{\mathrm{C}} nil; (N-1, S')}\text{[Eval-next-countdown-C]}$$

$$\frac{\Gamma; S \vdash E \xrightarrow{\mathrm{C}} nil; S'}{\Gamma; (0, S) \vdash \mathrm{next}\, E \xrightarrow{\mathrm{C}} nil; (0, S')}\text{[Eval-next-C]}$$

Id Identifiers are both in charge of looking up the value pointed to by I and buffering it by the appropriate amount. This is so that any downstream operators receive correctly synchronised values - ensuring the buffers are the correct size is one of the main responsibilities of the init phase.

As elsewhere, **buffer** is a shorthand for cycling the contents of the buffer b - b acts as a queue that is initially full of *nil* values, causing a delay as long as the buffer in the values yielded by this expression compared to the actual timing of the values at $\Gamma(I)$.

$$\frac{\begin{array}{c}\Gamma(I) = V \\ \mathbf{buffer}(b, V) = b', V_b\end{array}}{\Gamma, b \vdash I \to V_b; b'}\text{[Eval-id]}$$

Choke evaluation still cycles the buffer but drops the value the comes out of the buffer. This is in order to maintain the illusion that I is yielding values at a delay equal to buffer size - if instead we pushed a nil value onto the back of the queue, we would effectively be replacing "future" values with nils instead of the one that would be yielded at the current timestep.

$$\frac{\Gamma(I) = V \quad \mathbf{buffer}(b, V) = b', V_b}{\Gamma, b \vdash I \xrightarrow{\text{C}} nil; b'} [\text{Eval-id-C}]$$

3 Implementation

The implementation of Pixy presents a few specific challenges. For starters, to be able to determine the delays of variables, you need to be able to solve what amounts to a constraint solving problem. On top of that, you have to be able to properly line up the buffers, and ensure that all access happen seamlessly.

The implementation can be found at (`https://github.com/fhackett/pixy-lang`).

3.1 Constraint Solving

The constraint problem at hand is in the domain of Linear Integeger Arithmetic. This consists of linear equations like $x * 2 = y$ and $max(x, y) > 4$. To solve this, we implemented a simple constraint solver in Haskell, CLPHS (`https://github.com/TOTBWF/clphs`).

3.2 Buffering

Implementing buffering poses a few challenges. For efficiency reasons, we want to minimize data duplication. This means that having individual buffers for each variable reference is out of the question. Instead, we compute offsets for each variable reference, and use those to index into our buffers. For example, consider the code:

```
fib = 0 fby 1 fby (fib + next fib)
```

fib must have a 2-cell buffer,as references access it at 2 distinct points in time.

The basic sketch of the algorithm for computing the offsets is as follows:

1. Traverse the tree, and recursively build up a map of variables to delays.
2. Whenever you need to union together those maps, (For example, on a binop), take the larger delay.
3. Whenever you hit a where clause, union together all of the maps generated by the variable expressions, and then create buffers for each variable in the where clause.

- can the system afford to wait? What other components might be affected? It is worth noting that while the algorithm is simple in concept a lot of care needs to go into correctly calculating and combining variable delays - this algorithm is heavily context-based. That said, we have achieved decent results on our test cases with this setup.

4 Comments and future work

There is clearly a lot more to be done to the language. At the evaluation level we need to properly refine recursive operator application, and most of our ideas for a type system are discussed but not fully explored. While we have described some of these ideas inline, here we will present a set of possible future directions and our thoughts on them.

4.1 Circuits and lowest common denominator of computation

As we worked more on the language, we discovered that Pixy is not very powerful compared to most languages. In its current form it is unable to perform basic tasks we might expect from a general-purpose language like performing tasks that require unbounded memory or interacting with the world.

While on its own this can be seen as a weakness in the language, we noticed at the same time that due to its fully static control flow Pixy can be used to describe algorithms that would work on more unusual platforms such as field programmable gate arrays - since it derives from Lucid, which had both a text and a diagram representation, and removes most of Lucid's dynamism, Pixy should be able to express algorithms targeted at the lowest common denominator of computation: digital circuits. Since we also envisage interesting implementation strategies for commodity hardware, once we build a better base for the language it may be interesting to investigate what happens if you apply Pixy to circuit design.

4.2 State of the type system(s)

Current discussion places Pixy type system as 2-dimensional along the data and control axes. On the one hand we envisage a traditional type system similar to that used in typical functional programming languages that would be used to make sure basic data is coherent across all program timesteps, and on the other we hope to experiment with a temporal logic-based system that tries to reason about tricky corner cases and synchronisation problems that span multiple timesteps.

Part of our work on synchronising pre-delays might be generalisable to become a part of the temporal type system, and can already catch some basic programming errors such as leaving "gaps" in between the left and right operand of a fby.

One strategy we have in mind for implementing the temporal logic system is coinductive symbolic evaluation across different possible program states, taking advantage of the finite state space we describe in the init phase in order to argue termination of the checking process.

4.3 Expressing iteration

One notable missing feature from Lucid is support for nested iteration. Here we explore why this feature disappeared and how we expect to replace it.

In Lucid one would have been able to write something like this:

```
c until c eq N where
    N is current n;
    c = 1 fby c+1;
end
```

Given some external variable n, for each value of n this would yield the sequence $1 \ldots n-1$. While this is a reasonable program to write, within the context of Pixy's state machine-based semantics one might ask the question: if used within another program, when will these values be yielded? For this program, the question does not have an answer. Since for every timestep a new value of n may be presented, there can exist no state machine that will yield more than one value in a single timestep.

While this can be seen as a limitation of Pixy relative to Lucid's expressiveness, this also demonstrates that Pixy's state machine model enforces that each timestep of a Pixy program will take a known amount of time - it is impossible to write a program whose timesteps take an unbounded amount of time to execute.

The closest alternative we imagine is Turing-incomplete synchronous recursion, also referred to as recursive operator application earlier in this document. This would allow processing batches of data in one timestep, but using a mechanism more similar to C++'s templates. All calls would be expanded at compile time and the maximum recursion depth would be checked in order to ensure that a deadline can be given and met for the completion of each program timestep.

For truly infinite computation, the programmer would have to express it over multiple timesteps. This is reasonable, since Pixy specifically prioritises liveness and concurrency over expressiveness. It also encourages developers to think about the liveness of their systems.

4.4 Pixy's representation and treatment of numbers

In order to reason precisely about the symbolic evaluation of Pixy terms we may need to revisit and refine Pixy's model of numbers. Since we assume any value in Pixy will have a bounded size, it becomes important to define precisely how different operations interact with this size. What happens if you add 1 to the largest number a variable could represent? For sized types as described in [1] to work we also need to reason about lower bounds - what happens if you inductively subtract 2 from an unknown value? How do we deal with the possibility of reaching $1 - 2$ while unfolding a recursive operator application?

We suspect some of these issues will be clarified by implementing our planned explicit type system, but we can also see that the subtleties of some of these numerical edge cases may in turn inform what our type system should look like.

References

[1] Abel A. *MiniAgda: Integrating Sized and Dependent Types* Department of Computer Science, Ludwig-Maximilians-University Munich, Germany

[2] Wadge W. W., Ashcroft E. A. *Lucid, the Dataflow Programming Langugage* Academic Press, 1985