Pixy progress report

A. Finn Hackett, Reed Mullanix

April 23, 2018

Contents

1 Introduction

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

Our progress so far is that we have redesigned the language's evaluation semantics from the ground up to create Pixy, a language that shares a lot of syntax and surface-level semantics with Lucid. Pixy is however at its core a very different language.

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:

$\langle \mathit{expr} \rangle ::= \langle \mathit{number} \rangle$
$\quad | \ \langle \mathit{var} \rangle$
$\quad | \ \langle \mathit{bool} \rangle$
$\quad | \ \texttt{nil}$
$\quad | \ \texttt{?} \ \langle \mathit{expr} \rangle$
$\quad | \ \texttt{if} \ \langle \mathit{expr} \rangle \ \texttt{then} \ \langle \mathit{expr} \rangle \ \texttt{else} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{fby} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{where \{} \ \langle \mathit{wheredecls} \rangle \ \texttt{\}}$
$\quad | \ \texttt{next} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{var} \rangle \ \texttt{(} \ \langle \mathit{exprlist} \rangle \ \texttt{)}$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{+} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{-} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{*} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{expr} \rangle \ \texttt{/} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{exprtuple} \rangle$

$\langle \mathit{exprtuple} \rangle ::= \texttt{(} \ \langle \mathit{exprlist} \rangle \ (\ \texttt{|} \ \langle \mathit{expr} \rangle \ )? \ \texttt{)}$

$\langle \mathit{bool} \rangle ::= \texttt{true} \ | \ \texttt{false}$

$\langle \mathit{exprlist} \rangle ::= \langle \mathit{expr} \rangle \ \texttt{,} \ \langle \mathit{exprlist} \rangle \ | \ \langle \mathit{expr} \rangle$

$\langle \mathit{varlist} \rangle ::= \langle \mathit{var} \rangle \ \texttt{,} \ \langle \mathit{varlist} \rangle \ | \ \langle \mathit{var} \rangle$

$\langle \mathit{wheredecl} \rangle ::= \langle \mathit{var} \rangle \ \texttt{=} \ \langle \mathit{expr} \rangle$
$\quad | \ \langle \mathit{var} \rangle \ \texttt{(} \ \langle \mathit{varlist} \rangle \ \texttt{)} \ \texttt{=} \ \langle \mathit{expr} \rangle$
$\quad | \ \texttt{(} \ \langle \mathit{varlist} \rangle \ (\ \texttt{|} \ \langle \mathit{var} \rangle \ )? \ \texttt{)} \ \texttt{=} \ \langle \mathit{expr} \rangle$

$\langle \mathit{wheredecls} \rangle ::= \langle \mathit{wheredecl} \rangle \ \texttt{;} \ \langle \mathit{wheredecls} \rangle \ | \ \langle \mathit{wheredecl} \rangle$

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

We support 3 common datatypes: booleans, natural numbers and tuples.

2.1 Operational semantics

Pixy's operational semantics have several stages in order to model the ahead-of-time requirements of the language. This helps to prove by construction that Pixy's key properties hold.

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 was split off from the first stage due to complexity concerns.

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 lifecycle of the program and repeats indefinitely.

2.1.1 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 [], 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 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 excatly 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.

2.1.2 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.

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

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

Following the basic semantics, at t_1 x will be 1, and when computing y we will take 1 and try to add it to $\text{next}\,x$, which dropped the value 1 and is equal to *nil* in this case. This is counter-intuitive since $x + \text{next}\,x$ looks like it means "wait for $\text{next}\,x$" 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.

The syntax for derivations of this stage looks like this:

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

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.

In order to perform delay compensation, some initial state will use a notation $Q[x]$ as a shorthand for creating a queue of length x. This queue will initially be empty, and should be filled at maximum to capacity x.

$$\frac{\begin{array}{c} d \vdash A \overset{\text{init}}{\Rightarrow} S_a; \delta_a; \Delta_a \\ d \vdash B \overset{\text{init}}{\Rightarrow} S_b; \delta_b; \Delta_b \end{array}}{d \vdash A\ op\ B \overset{\text{init}}{\Rightarrow} (S_a, Q[\max(\delta_a, \delta_b) - \delta_a], S_b, Q[\max(\delta_a, \delta_b) - \delta_b]); \max(\delta_a, \delta_b); \Delta_a, \Delta_b} \text{[Init-binop]}$$

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

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

$$\frac{}{d \vdash I \overset{\text{init}}{\Rightarrow} (); \delta(I);} \text{[Init-id]}$$

$$\frac{\begin{array}{c} \dfrac{}{d \vdash V \overset{\text{init}}{\Rightarrow} S_v; \delta_v; \Delta_v} V... \\ d \vdash E \overset{\text{init}}{\Rightarrow} S; \delta; \Delta \end{array}}{d \vdash [A = V...]E \overset{\text{init}}{\Rightarrow} (S, (S_v...), (Q[\max(\delta_v...) - \delta_v]...)); \delta; \Delta, \Delta_v..., \delta(A) = \delta_v...} \text{[Init-operator-scope]}$$

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

2.1.3 Proof of δ correctness

TODO

2.1.4 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: $\Gamma; S \vdash E \Downarrow V; S'$.

Conditionals We haven't considered if statements in great detail so far - they pose a peculiar problem. Unlike typical programming languages, if statements in Pixy do not short-circuit. That is, we must ensure that concurrent events continue uninterrupted in both the taken and non-taken branches of the if statement.

To achieve this we introduce a second mode of evaluation, choke-evaluation. It is notated $\not\Downarrow$ and aside from its special implementation acts the same as normal evaluation. Specifically, if an expression if choke-evaluated then it should not appear to perform any computations. It may however update any of its non-visible state as normal.

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

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

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

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

Binop This is a catch-all derivation for any binary operator in Pixy - the main point of interest here is that binary operators synchronise their inputs using buffers allocated in the init stage.

$\mathbf{buffer}(Q, V)$ in this case is a shorthand for pushing V onto the queue Q, returning the element at the front of the queue is any, nil otherwise.

$$\frac{\begin{array}{c}\Gamma; S_l \vdash L \Downarrow V_l; S'_l \\ \Gamma; S_r \vdash R \Downarrow V_r; S'_r \\ \mathbf{buffer}(Q_l, V_l) = Q'_l, D_l \\ \mathbf{buffer}(Q_r, V_r) = Q'_r, D_r\end{array}}{\Gamma; (S_l, Q_l, S_r, Q_r) \vdash L\ op\ R \Downarrow D_l\ op\ D_r; (S'_l, Q'_l, S'_r, Q'_r)}\,[\text{Eval-binop}]$$

$$\frac{\begin{array}{c}\Gamma; S_l \vdash L \mathrel{\Downarrow\!\!\!\!\!\!-\,} nil; S'_l \\ \Gamma; S_r \vdash R \mathrel{\Downarrow\!\!\!\!\!\!-\,} nil; S'_r \\ \mathbf{buffer}(Q_l, nil) = Q'_l, D_l \\ \mathbf{buffer}(Q_r, nil) = Q'_r, D_r\end{array}}{\Gamma; (S_l, Q_l, S_r, Q_r) \vdash L\ op\ R \mathrel{\Downarrow\!\!\!\!\!\!-\,} D_l\ op\ D_r; (S'_l, Q'_l, S'_r, Q'_r)}\,[\text{Eval-binop-C}]$$

Where

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

An interesting observation here is that even under choke-evaluation where expression still evaluate their internal state. This is because the where's internal state is not directly visible - hidden computation can occur as long as it is not returned from the main body of the where.

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

Operator scope Even though operator application will have been flattened in stage 1, we still need to apply the proper scoping rules and synchronisation here.

Specifically, we ensure all the arguments have their pre-delays synchronised and that the variable scope contains only the arguments to the operator.

$$\dfrac{\begin{array}{c}\overline{\Gamma; S_v \vdash E_v \Downarrow V_v; S'_v}^{\,E_v\ldots}\\ \overline{\mathbf{buffer}(Q_v, V_v) = Q'_v, D_v}^{\,Q_v\ldots}\\ A = D_v\ldots; S \vdash E \Downarrow V; S'\end{array}}{\Gamma; (S, (S_v\ldots), (Q_v\ldots)) \vdash [A = E_v\ldots]E \Downarrow V; (S', (S'_v\ldots), (Q'_v\ldots))}\text{[Eval-apply]}$$

$$\dfrac{\begin{array}{c}\overline{\Gamma; S_v \vdash E_v \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'_v}^{\,E_v\ldots}\\ \overline{\mathbf{buffer}(Q_v, nil) = Q'_v, D_v}^{\,Q_v\ldots}\\ A = D_v\ldots; S \vdash E \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'\end{array}}{\Gamma; (S, (S_v\ldots), (Q_v\ldots)) \vdash [A = E_v\ldots]E \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; (S', (S'_v\ldots), (Q'_v\ldots))}\text{[Eval-apply-C]}$$

fby

$$\dfrac{\begin{array}{c}\Gamma; S_a \vdash A \Downarrow V_a; S'_a\\ \Gamma; S_b \vdash B \Downarrow V_b; S'_b\\ V_b \neq nil\end{array}}{\Gamma; (false, U, S_a, S_b) \vdash A \,\mathrm{fby}\, B \Downarrow V_a; (V_a \neq nil, V_b, S'_a, S'_b)}\text{[Eval-fby-A-buffer]}$$

$$\dfrac{\begin{array}{c}\Gamma; S_a \vdash A \Downarrow V_a; S'_a\\ \Gamma; S_b \vdash B \Downarrow V_b; S'_b\\ V_b = nil\end{array}}{\Gamma; (false, U, S_a, S_b) \vdash A \,\mathrm{fby}\, B \Downarrow V_a; (V_a \neq nil, U, S'_a, S'_b)}\text{[Eval-fby-A-nobuffer]}$$

$$\dfrac{\begin{array}{c}\Gamma; S_a \vdash A \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'_a\\ \Gamma; S_b \vdash B \Downarrow V_b; S'_b\\ U \neq nil\end{array}}{\Gamma; (true, U, S_a, S_b) \vdash A \,\mathrm{fby}\, B \Downarrow U; (true, V_b, S'_a, S'_b)}\text{[Eval-fby-B-buffer]}$$

$$\dfrac{\begin{array}{c}\Gamma; S_a \vdash A \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'_a\\ \Gamma; S_b \vdash B \Downarrow V_b; S'_b\\ U = nil\end{array}}{\Gamma; (true, U, S_a, S_b) \vdash A \,\mathrm{fby}\, B \Downarrow V_b; (true, nil, S'_a, S'_b)}\text{[Eval-fby-B-nobuffer]}$$

$$\dfrac{\begin{array}{c}\Gamma; S_a \vdash A \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'_a\\ \Gamma; S_b \vdash B \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; S'_b\end{array}}{\Gamma; (C, U, S_a, S_b) \vdash A \,\mathrm{fby}\, B \mathrel{\mathrm{C}\!\!\!\!\Downarrow} nil; (C, U, S'_a, S'_b)}\text{[Eval-fby-C]}$$

next

$$\frac{\begin{array}{c}\Gamma; S \vdash E \Downarrow V; S' \\ V = nil\end{array}}{\Gamma; (false, S) \vdash \text{next}\, E \Downarrow nil; (false, S')}\text{[Eval-next-skip-nil]}$$

$$\frac{\begin{array}{c}\Gamma; S \vdash E \Downarrow V; S' \\ V \neq nil\end{array}}{\Gamma; (false, S) \vdash \text{next}\, E \Downarrow nil; (true, S')}\text{[Eval-next-skip-first]}$$

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

$$\frac{\Gamma; S \vdash E \mathrel{\Downarrow\mkern-14mu\mathrm{C}} nil; S'}{\Gamma; (C, S) \vdash \text{next}\, E \mathrel{\Downarrow\mkern-14mu\mathrm{C}} nil; (C, S')}\text{[Eval-next-C]}$$

Id

$$\frac{\Gamma(I) = V}{\Gamma, () \vdash I \Downarrow V; ()}\text{[Eval-id]}$$

$$\frac{}{\Gamma, () \vdash I \mathrel{\Downarrow\mkern-14mu\mathrm{C}} nil; ()}\text{[Eval-id-C]}$$

3 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.

3.1 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 isn't even Turing complete.

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 dynamicism, 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.

3.2 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.

3.3 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 - can the system afford to wait? What other components might be affected?