INDEXICAL TRANSLATION OF
TAIL-RECURSIVE FUNCTIONS

JEAN-RAYMOND GAGNÉ
Département d’informatique, Université Laval, Québec (Québec) Canada G1K 7P4
E-mail: gagne@ift.ulaval.ca

JOEY PAQUET
Department of Computer Science, Concordia University, Montréal (Québec) Canada H3G 1M8
E-mail: paquet@cs.concordia.ca

JOHN PLAICE
School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia
E-mail: plaice@cse.unsw.edu.au

We show that a very general form of Lucid (and RLUCID) tail-recursive function can be transformed into an indexical equivalent. We show also that the standard indexical translations of the \texttt{wvr} and \texttt{upon} functions can be considered to be particular cases of the general situation. We give full proofs of the results, taking advantage of the clean semantics of Lucid.

1 Introduction

With the rapid advances in the semantics of the Lucid family of languages,\(^1,2,3\) it is clear that there is a need for really efficient implementations of Lucid. There are many levels at which efficiency can be sought.

Since Lucid focuses on iteration, it is desirable to eliminate all uses of recursive functions, and replace them with indexical equivalents. The seminal work by Panagiotis Rondogiannis and William Wadge on higher-order functions\(^5,6\) has significantly advanced in the compilation of functions in which the recursion is required to compute values.

But recursion also appears in Lucid to define functions that will act over the successive elements of its input streams. Typically, these functions are tail-recursive, consuming elements at the head of a stream as needed. In this article, we examine a general class of tail-recursive functions, and show that they can be translated into non-recursive functions, using the basic Lucid operators. Such translations have already given for the \texttt{wvr} and \texttt{upon} operators\(^1\) (page 253), but no formal proof of the translation was ever given.
The article is arranged as follows. Section 2 will present the class of tail-recursive functions that we will be examining. Section 3 will show how these functions can be translated into Lucid functions using only \texttt{wvr} and \texttt{skip}, the generalized \texttt{next} operator; this work is inspired from Gagné’s Ph.D. thesis.\textsuperscript{4} Section 4 gives the formal proof of the translations of \texttt{wvr} and \texttt{skip} into expressions using only \texttt{@} and \texttt{#}; this work is inspired from Paquet’s Ph.D. thesis.\textsuperscript{2}

## 2 A class of tail-recursive functions

Before introducing the class of tail-recursive functions that is of interest, we first introduce some notation. We begin by defining the permutation of a tuple of dataflows.

**Definition 1** Let \( \vec{x} = \langle x_1, \ldots, x_n \rangle \) be an \( n \)-tuple of dataflows, and let \( \vec{v} = \langle v_1, \ldots, v_n \rangle \) be an \( n \)-tuple of integers in \( 1..n \), each appearing only once. Then \( \vec{v} \odot \vec{x} \) gives a new \( n \)-tuple of dataflows, a permutation of \( \vec{x} \), defined by:

\[
\vec{v} \odot \vec{x} = \langle x_{v_1}, \ldots, x_{v_n} \rangle.
\]

We will write \( \odot^{-1} \) for the left inverse of \( \odot \), i.e.

\[
\vec{x} = \vec{v} \odot^{-1} (\vec{v} \odot \vec{x}).
\]

We define the operators \texttt{wvr} and \texttt{skip}.

**Definition 2**

\[
X \texttt{wvr} Y \overset{\text{def}}{=} \begin{cases} 
\text{if first } Y \text{ then } X \texttt{fby} (\text{next } X \texttt{wvr} \text{ next } Y) \\
\text{else } (\text{next } X \texttt{wvr} \text{ next } Y)
\end{cases} \tag{2.1}
\]

\[
X \texttt{skip} N \overset{\text{def}}{=} \begin{cases} 
\text{if first } N = 0 \text{ then } X \texttt{fby} (X \texttt{skip} (\text{next } N)) \\
\text{else } (\text{next } X) \texttt{skip} ((N - 1) \texttt{fby} (\text{next } N))
\end{cases} \tag{2.2}
\]

The definitions for \texttt{skip} and \texttt{wvr} are extrapolated to tuples of dataflows.

**Definition 3** Let \( \vec{x} = \langle x_1, \ldots, x_n \rangle \) be a tuple of dataflows, \( \vec{v} = \langle v_1, \ldots, v_n \rangle \) be an \( n \)-tuple of integer dataflows, and \( \vec{b} = \langle b_1, \ldots, b_n \rangle \) be an \( n \)-tuple of Boolean dataflows. Then

\[
\vec{x} \texttt{skip} \vec{v} = \langle x_1 \texttt{skip} v_1, \ldots, x_n \texttt{skip} v_n \rangle
\]

\[
\vec{x} \texttt{wvr} \vec{b} = \langle x_1 \texttt{wvr} b_1, \ldots, x_n \texttt{wvr} b_n \rangle
\]

We will also be using the sequential version of Dijkstra’s guarded command.
Definition 4 Let $C_1, \ldots, C_m$ be Boolean expressions and let $E_1, \ldots, E_m$ be expressions. Then
\[
\left( \bigwedge_{i=1..m} C_i \rightarrow E_i \right) = \left( \text{if } C_1 \text{ then } E_1 \text{ else if } C_2 \text{ then } E_2 \ldots \text{ else } E_m \right)
\]

Definition 5 Let $\vec{c} = \langle c_1, \ldots, c_p \rangle$ be a $p$-tuple of constants and let $\vec{x} = \langle x_1, \ldots, x_q \rangle$ be a $q$-tuple of dataflows. Then a well-formed Lucid expression $E(\vec{c}, \vec{x})$ has as nullary elements the constants of the basic algebra, along with occurrences of the constants $c_a$, $a = 1..p$, and specific elements of the $x_b$, $b = 1..q$, i.e. elements of the form first $x_b$; and whose non-nullary arguments are all data operators.

We are now ready for the definition of a well-formed recursive function.

Definition 6 A well-formed recursive Lucid function is of the form:
\[
F(\vec{c}, \vec{x}) = \bigwedge_{i=1..m} \alpha_i(\vec{c}, \vec{x}) \rightarrow \beta_{\ell_i}(\vec{c}, \vec{x}) \text{ fby } \ldots \text{ fby } \beta_{\ell_i}(\vec{c}, \vec{x}) \text{ fby } F(\vec{c}_i, \vec{x}_i)
\]

where

- $\vec{c} = \langle c_1, \ldots, c_p \rangle$ is a $p$-tuple of constants;
- $\vec{x} = \langle x_1, \ldots, x_q \rangle$ is a $q$-tuple of input dataflows;
- for all $i = 1..m$, $\alpha_i(\vec{c}, \vec{x})$ is a well-formed Boolean expression;
- for all $i = 1..m$ and $j = 1..\ell_i$, $\beta_{ij}(\vec{c}, \vec{x})$ is a well-formed expression;
- for all $i = 1..m$, $\vec{c}_i = \langle \gamma_{i1}(\vec{c}, \vec{x}), \ldots, \gamma_{ip}(\vec{c}, \vec{x}) \rangle$ is a $p$-tuple of well-formed expressions;
- for all $i = 1..m$, $\vec{x}_i = (\tau_i \oplus \vec{x}) \text{ skip } \nu_i$ is a permutation of the inputs, advancing some of the components with a fixed number of next operators; hence $\tau_i$ is a permutation of $(1, \ldots, q)$ and $\nu_i$ is a $q$-tuple of integers.

3 Compiling out tail-recursion

The indexical translation consists of determining which iteration is currently running, i.e. how many times the function has recursed. This allows the computation of the appropriate indices for each of the arguments. Once this is done, then we need to know which element of the current iteration must be produced, and everything can be computed.

This general translation approach works smoothly, except in the situation where one of the conditions might not provoke any output. This is the case,
for example, for the \texttt{wvr} function, which produces no output if the Boolean condition is false. We deal with such situations in two stages. First, we create a function that generates a special empty value, written \texttt{1}. Then we filter out these situations with the \texttt{wvr} operator. The \texttt{wvr} operator itself requires a separate transformation, explained in detail in the next section.

Consider now function $F$ of Definition 6. It can be rewritten into an iterative function $F'$ using the following rule:

$$F'(c, x) = r \texttt{ wvr } r'$$

where

$$\langle r', r \rangle = \left[ \begin{array}{c}
\alpha_i(\kappa, \bar{y}) \\
i = 1 \ldots m
\end{array} \right] \rightarrow \left[ s > \ell_i \rightarrow \langle \text{false}, 0 \rangle \\
 s \leq \ell_i \rightarrow \left[ \begin{array}{c}
\text{true, } \\
j = 1 \ldots \ell_i
\end{array} \right] s = j \rightarrow \beta_{ij}(\kappa, \bar{y}) \right]$$

$$\langle s, v, \tau, \kappa \rangle = \langle 1, \bar{0}_p, \bar{w}, \bar{c} \rangle \texttt{ fby }$$

$$\left[ \begin{array}{c}
\alpha_i(\kappa, \bar{y}) \\
i = 1 \ldots m
\end{array} \right] \rightarrow \left[ s \leq \ell_i \rightarrow \langle s + 1, \bar{0}_p, \tau, \kappa \rangle \\
 s > \ell_i \rightarrow \langle 1, (\tau_i \ominus \tau') \ominus v_i, \tau_i \ominus \tau, \kappa' \rangle \right]$$

$$\ell_i = \ell(\beta_i)$$

$$\kappa_i = \langle \gamma_1(\kappa, \bar{y}), \ldots, \gamma_m(\kappa, \bar{y}) \rangle$$

$$\bar{y} = \tau \ominus (\bar{x} \texttt{ skip } (\tau \ominus^{-1} v))$$

$$\bar{w} = (1, \ldots, p)$$

$$\bar{0}_p = \langle 0, \ldots, 0 \rangle$$

Applying this approach to

first $X = X \texttt{ fby } \text{first } X$

yields:

first $X = R$

where

$$R = r \texttt{ wvr } r'$$

$$(r', r) = \texttt{ if } s > 1 \texttt{ then } \langle \text{false}, 0 \rangle$$

else $\langle \text{true}, y_0 \rangle$

$$(s, v, t, k) = \langle 1, <0>, <0>, <> \rangle \texttt{ fby }$$

if $s < 2$ then $(s + 1, <0>, <0>, <>)$

else $\langle 1, <1>, <0>, <> \rangle$

$y_0 = X \texttt{ skip } \text{select}(1, v)$$
After removing unused dataflows, we get:

```plaintext
first X = R
    R = y0 wvr s=1
    y0 = X skip (if s=2 then 1 else 0)
    s  = 1 fby (if s<2 then 2 else 1)
which is equivalent to:
    first X = X skip 0
```

Applying this approach to the RLUCID definition:

```plaintext
current(A, B, C) =
    choose
    A: current(next A, B, first A)
    B: C fby current(A, next B, first C)
end
```
yields

```plaintext
current(A, B, C) =
    R = r wvr r'
    (r',r) = if (y0 before y1) then (false, 0)
              else (true, select(1,k))
    (s,v,t,k) = (1, <0,0>, <0,1>, <C>) fby
              if (y0 before y1) then (1, <1,0>, <0,1>, <y0>)
                 else if (s<2) then (s+1, <0,0>, <0,1>, k)
                 else (1, <0,1>, <0,1>, k)
    y0 = A skip select(1,v)
    y1 = B skip select(2,v)
```

After removing unused dataflows and some rewriting, we get:

```plaintext
current(A, B, C) =
    R = k wvr not(y0 before y1)
    (s,v,k) = (1, <0,0>, C) fby
               if (y0 before y1) then (1, <1,0>, y0)
               else if (s<2) then (s+1, <0,0>, k)
               else (1, <0,1>, k)
    y0 = A skip select(1,v)
    y1 = B skip select(2,v)
```

It should be noted that it is easy to rewrite nested tail-recursive functions into a unique tail recursive function. This can be done by assigning a unique number to each of the functions and by adding an extra parameter to each function, along with a test that guards the execution of the function's body. The parameter lists of the functions can then be normalized so that they can be merged, and finally the body of all of the functions can be merged.
4 Compiling out selection

In this section, we will prove that the tail-recursive and indexical definitions for the \texttt{wfr} operator are equivalent. The point of this section is not just to make this proof, but also to show how exact proofs can be undertaken using Lucid. We begin by introducing some notation.

**Definition 7** Let $X = (x_0, x_1, \ldots, x_i, \ldots)$ and $Y = (y_0, y_1, \ldots, y_i, \ldots)$ be two pipeline dataflows. We will write $[X]_i$ for $x_i$ and $X^i$ for the stream $X$ advanced by $i$ positions:

$$X^i = (x_i, x_{i+1}, \ldots, x_{i+j}, \ldots)$$

We will also write $X = Y$ iff $(\forall i : i \geq 0 : x_i = y_i)$.

We now give an axiomatization of Lucid. We make no pretense that it is complete.

**Axiom 8** Let $i \geq 0$.

\begin{align*}
[c]_i &= c \quad (8.1) \\
[X + c]_i &= [X]_i + c \quad (8.2) \\
[\text{first } X]_i &= [X]_0 \quad (8.3) \\
[\text{next } X]_i &= [X]_{i+1} \quad (8.4) \\
[X \overline{fby} Y]_0 &= [X]_0 \quad (8.5) \\
[X \overline{fby} Y]_{i+1} &= [Y]_i \quad (8.6) \\
\text{if true then } [X]_i \text{ else } [Y]_i &= [X]_i \quad (8.7) \\
\text{if false then } [X]_i \text{ else } [Y]_i &= [Y]_i \quad (8.8) \\
\text{if } C \text{ then } X \text{ else } Y)_i &= \text{if } C)_i \text{ then } [X]_i \text{ else } [Y]_i \quad (8.9) \\
X^0 &= X \quad (8.10) \\
[X^i]_0 &= [X]_i \quad (8.11) \\
\text{first } X^i &= [X]_i \quad (8.12) \\
\text{next } X^i &= X^{i+1} \quad (8.13) \\
\text{next } (X \overline{fby} Y) &= Y \quad (8.14) \\
(\text{first } X) \overline{fby} Y &= X \overline{fby} Y \quad (8.15) \\
\text{if true then } X \text{ else } Y &= X \quad (8.16) \\
\text{if false then } X \text{ else } Y &= Y \quad (8.17) 
\end{align*}

We also define the derived operator \texttt{wfr}.
Definition 9

\[ X \text{ wpr } Y \overset{\text{def}}{=} \text{if } \text{first} \ Y \text{ then } X \text{ fby } (\text{next } X \text{ wpr } \text{next} \ Y) \]
\[ \text{else } (\text{next } X \text{ wpr } \text{next} \ Y) \]

(9.1)

It turns out that these primitives are not the easiest to work with. By taking a different approach, it is possible to have random access into streams, using an index \# corresponding to the current position. No longer are we manipulating infinite extensions (streams), rather we are defining computation according to a context (here a single integer). Everything is now redefined in terms of the operators \# and \@. In terms of the three original operators, these are defined as:

Definition 10

\[ \# \overset{\text{def}}{=} 0 \text{ fby } (\# + 1) \]

(10.1)

\[ X @ Y \overset{\text{def}}{=} \text{if } Y = 0 \text{ then } \text{first} \ X \]
\[ \text{else } (\text{next } X) @ (Y - 1) \]

(10.2)

Below, we will give definitions for the original operators using these two new operators. We first show some basic properties of \@ and \#.

Proposition 11 \((\forall i : i \geq 0 : [\#]_i = i)\).

Proof Proof by induction over \(i\).

Base step \((i = 0)\).

\[ [\#]_0 = [0 \text{ fby } (\# + 1)]_0 \]
\[ = [0]_0 \]
\[ = 0 \]

(10.1)

Induction step \((i = k + 1)\).

Suppose \((\forall i : 0 \leq i \leq k : [\#]_i = i)\).

\[ [\#]_{k+1} = [0 \text{ fby } (\# + 1)]_{k+1} \]
\[ = [\# + 1]_k \]
\[ = [\#]_k + 1 \]
\[ = k + 1 \]

(IH)

Hence \((\forall i : i \geq 0 : [\#]_i = i)\).

\(\Box\)

Proposition 12 \((\forall i : i \geq 0 : [Y]_i \Rightarrow ([X @ Y]_i = [X[Y]_i])\).

Proof Let \(i \geq 0\). We will prove by induction over \([Y]_i\) that \([Y]_i \geq 0 \Rightarrow [X @ Y]_i = [X[Y]_i].\)
Base step \((y_i = 0)\).
\[
[X @ Y]_i = \begin{cases} 
\text{if } Y = 0 \text{ then } \text{first } X \\
\text{else } (\text{next } X) @ (Y - 1) \end{cases} 
\]
\[= \begin{cases} 
\text{if } Y = 0; \text{ then } [\text{first } X]_i \\
\text{else } [(\text{next } X) @ (Y - 1)]_i 
\end{cases} \tag{8.9}
\]
\[= \begin{cases} 
\text{if } Y; \text{ then } [\text{first } X]_i \\
\text{else } [(\text{next } X) @ (Y - 1)]_i 
\end{cases} \tag{8.2}
\]
\[= [\text{first } X]_i \tag{8.7}
\]
\[= [X]_0 \tag{8.3}
\]
\[= [X]_{[Y]_i} \tag{IH}
\]

Induction step \((y_i = k + 1)\).
Suppose \((\forall i : 0 \leq i \leq k : [Y]_i \Rightarrow ([X @ Y]_i = [X]_{[Y]_i}))\).
\[
[X @ Y]_i = \begin{cases} 
\text{if } Y = 0 \text{ then } \text{first } X \text{ else } (\text{next } X) @ (Y - 1) \end{cases} \tag{10.2}
\]
\[= \begin{cases} 
\text{if } Y = 0; \text{ then } [\text{first } X]_i \\
\text{else } [(\text{next } X) @ (Y - 1)]_i 
\end{cases} \tag{8.9}
\]
\[= \begin{cases} 
\text{if } Y; \text{ then } [\text{first } X]_i \\
\text{else } [(\text{next } X) @ (Y - 1)]_i 
\end{cases} \tag{8.2}
\]
\[= [(\text{next } X) @ (Y - 1)]_i \tag{8.8}
\]
\[= [\text{next } X]_{[Y]_{i-1}} \tag{IH}
\]
\[= [\text{next } X]_{[Y]_{i-1}} \tag{8.2}
\]
\[= [X]_{[Y]_{i-1} + 1} \tag{8.4}
\]
\[= [X]_{[Y]_i} \tag{()}\]

Hence \((\forall i : i \geq 0 : [Y]_i \geq 0 \Rightarrow ([X @ Y]_i = [X]_{[Y]_i}))\). \quad \square

Definition 13 Here are the new definitions of the basic operators.

\[
\text{first } X \overset{\text{def}}{=} X @ 0 \tag{13.1}
\]
\[
\text{next } X \overset{\text{def}}{=} X @ (\# + 1) \tag{13.2}
\]
\[
X \text{ fby } Y \overset{\text{def}}{=} \begin{cases} 
\text{if } \# = 0 \text{ then } X \text{ else } Y @ (\# - 1) 
\end{cases} \tag{13.3}
\]
\[
X \text{ wrv } Y \overset{\text{def}}{=} X @ T \tag{13.4}
\]
where
\[
T = U \text{ fby } U @ (T + 1) \tag{13.5}
\]
\[
U = \begin{cases} 
\text{if } Y \text{ then } \# \text{ else next } U 
\end{cases} \tag{13.6}
\]

end
The advantage of these new definitions is that they do not use any form of recursive function definitions. Rather, all of the definitions are iterative, and in practice, more easily implemented in an efficient manner. We prove below that the new definitions are equivalent to the (underlined) old ones.

**Proposition 14** \( \text{first} \; X = \text{first} \; X \).

**Proof** Let \( i \geq 0 \). Then

\[
\begin{align*}
[\text{first} \; X]_i &= [X \; @ \; 0]_i \\
&= [X]_{0}i \\
&= [X]_{0} \\
&= [\text{first} \; X]_i
\end{align*}
\]  \hspace{1cm} (13.1)

Hence \( \forall i : i \geq 0 : [\text{first} \; X]_i = [\text{first} \; X]_i \). Hence \( \text{first} \; X = \text{first} \; X \). \( \Box \)

**Proposition 15** \( \text{next} \; X = \text{next} \; X \).

**Proof** Let \( i \geq 0 \). Then

\[
\begin{align*}
[\text{next} \; X]_i &= [X \; @ \; (# + 1)]_i \\
&= [X]_{(# + 1)}_i \\
&= [X]_{#i+1} \\
&= [X]_{i+1} \\
&= [\text{next} \; X]_i
\end{align*}
\]  \hspace{1cm} (13.2)

Hence \( \forall i : i \geq 0 : [\text{next} \; X]_i = [\text{next} \; X]_i \). Hence \( \text{next} \; X = \text{next} \; X \). \( \Box \)

**Proposition 16** \( \text{fby} \; Y = \text{fby} \; Y \).

**Proof** Proof by induction over \( i \).

**Base step** \( (i = 0) \).

\[
\begin{align*}
[X \; \text{fby} \; Y]_0 &= [\text{if} \; # = 0 \; \text{then} \; X \; \text{else} \; Y \; @ \; (# - 1)]_0 \\
&= \text{if} \; [\#]_0 = 0 \; \text{then} \; [X]_0 \; \text{else} \; [Y \; @ \; (# - 1)]_0 \\
&= \text{if} \; [\#]_0 = 0 \; \text{then} \; [X]_0 \; \text{else} \; [Y \; @ \; (# - 1)]_0 \\
&= \text{if} \; 0 = 0 \; \text{then} \; [X]_0 \; \text{else} \; [Y \; @ \; (# - 1)]_0 \\
&= [X]_0 \\
&= [X \; \text{fby} \; Y]_0
\end{align*}
\]  \hspace{1cm} (13.3)
Induction step \((i = k + 1)\).

\[
[X \text{ fby } Y]_{k+1} = [\text{if } \# = 0 \text{ then } X \text{ else } Y \text{ @ } (\# - 1)]_{k+1}
\]

\[
= \text{if } [\# = 0]_{k+1} \text{ then } [X]_{k+1} \text{ else } [Y \text{ @ } (\# - 1)]_{k+1}
\]

\[
= \text{if } [\#]_{k+1} = 0 \text{ then } [X]_{k+1} \text{ else } [Y \text{ @ } (\# - 1)]_{k+1}
\]

\[
= \text{if } k + 1 = 0 \text{ then } [X]_{k+1} \text{ else } [Y \text{ @ } (\# - 1)]_{k+1}
\]

\[
= [Y \text{ @ } (\# - 1)]_{k+1}
\]

\[
= [Y]_{\#-1, k+1}
\]

\[
= [Y]_{\#, k+1 - 1}
\]

\[
= [Y]_k
\]

\[
= [X \text{ fby } Y]_{k+1}
\]

Hence \((\forall i : i \geq 0 : [X \text{ fby } Y]_i = [X \text{ fby } Y]_i)\). Hence \text{ fby } = \text{ fby }.

The proof for \text{ wfr} is more complicated, as it requires relating an iterative definition to a recursive definition. We will therefore need four lemmas that refer to variables \(T\) and \(U\) in Definitions 13.5 and 13.6. In addition, we must define the \textit{rank} of a Boolean stream.

\textbf{Definition 17} Let \(Y\) be a Boolean stream.

\[
\text{rank}(-1, Y) \overset{\text{def}}{=} -1
\]

\[
\text{rank}(i+1, Y) \overset{\text{def}}{=} \min \{ k : k > \text{rank}(i, Y) : [Y]_k = \text{true} \}
\]

\textit{Below, we write \(r_i\) for rank\((i, Y)\).}

\textbf{Lemma 18} \((\forall i : i \geq 0 : (\forall j : r_{i-1} < j \leq r_i : X^j \text{ wfr } Y^j = X^{r_i \text{ wfr } Y^{r_i}}))\).

\textbf{Proof} Let \(i \geq 0\). Proof by downwards induction over \(j\). Note that \(r_{i-1} < r_i\).

\textbf{Base step} \((j = r_i)\).

\[
X^{r_i \text{ wfr } Y^{r_i}} = X^{r_i \text{ wfr } Y^{r_i}}
\]

\text{(Id)}
Induction step \((j = k - 1, j > r_{i-1})\).
Suppose \((\forall j : k \leq j \leq r_i : X^j \text{ wfr } Y^j = X^{r_i} \text{ wfr } Y^{r_i})\).

\[
X^{k-1} \text{ wfr } Y^{k-1} = \begin{cases} \text{if } \text{first } Y^{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wfr } Y^k \\ \text{else } X^k \text{ wfr } Y^k \end{cases} = \begin{cases} \text{if } [Y]_{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wfr } Y^k \\ \text{else } X^k \text{ wfr } Y^k \end{cases} = X^k \text{ wfr } Y^k = X^{r_i} \text{ wfr } Y^{r_i} \tag{9.1} = X^{r_i} \text{ wfr } Y^{r_i} \tag{IH} \]

Hence, \((\forall i : i \geq 0 : (\forall j : r_{i-1} < j \leq r_i : X^j \text{ wfr } Y^j = X^{r_i} \text{ wfr } Y^{r_i}))\). \qed

**Lemma 19** \((\forall i : i \geq 0 : (X \text{ wfr } Y)^i = X^{r_i} \text{ wfr } Y^{r_i})\).

**Proof** Proof by induction over \(i\).

**Base step** \((i = 0)\).

\[
(X \text{ wfr } Y)^0 = X \text{ wfr } Y = X^0 \text{ wfr } Y^0 = X^{r_0} \text{ wfr } Y^{r_0} \tag{18} \]

**Induction step** \((i = k + 1)\).
Suppose \((\forall i : 0 \leq i \leq k : (X \text{ wfr } Y)^i = X^{r_i} \text{ wfr } Y^{r_i})\).

\[
(X \text{ wfr } Y)^{k+1} = \text{next } ((X \text{ wfr } Y)^k) = \text{next } (X^{r_k} \text{ wfr } Y^{r_k}) = \begin{cases} \text{if } \text{first } Y^{r_k} \\ \text{then } X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1} \\ \text{else } X^{r_k+1} \text{ wfr } Y^{r_k+1} \end{cases} = \begin{cases} \text{if } [Y]_{r_k} \\ \text{then } X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1} \\ \text{else } X^{r_k+1} \text{ wfr } Y^{r_k+1} \end{cases} = \begin{cases} \text{next } (X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1}) \\ \text{next } (X^{r_k} \text{ wfr } Y^{r_k+1}) \end{cases} = X^{r_k+1} \text{ wfr } Y^{r_k+1} = X^{r_{i+1}} \text{ wfr } Y^{r_{i+1}} \tag{18} \]

Hence, \((\forall i : i \geq 0 : (X \text{ wfr } Y)^i = X^{r_i} \text{ wfr } Y^{r_i})\). \qed

**Lemma 20** \((\forall i : i \geq 0 : (\forall j : r_{i-1} < j \leq r_i : [U]_j = r_i))\).

**Proof** Let \(i \geq 0\). Proof by downwards induction over \(j\). Note that \(r_{i-1} < r_i\).
Base step ($j = r_i$).

\[ [U]_{r_i} = [\text{if } Y \text{ then } \#$ \text{ else next } U]_{r_i} \]
\[ = \text{if } [Y]_{r_i} \text{ then } [\#]_{r_i} \text{ else next } U]_{r_i} \]
\[ = [\#]_{r_i} \]
\[ = r_i \]  

(13.6)

(8.9)

(8.7)

(11)

Induction step ($j = k - 1$, $j > r_{i-1}$).

Suppose ($\forall j : k \leq j \leq r_i : [U]_j = r_i$).

\[ [U]_{k-1} = [\text{if } Y \text{ then } \#$ \text{ else next } U]_{k-1} \]
\[ = \text{if } [Y]_{k-1} \text{ then } [\#]_{k-1} \text{ else next } U]_{k-1} \]
\[ = [\text{next } U]_{k-1} \]
\[ = [U]_k \]
\[ = r_i \]  

(13.6)

(8.9)

(8.8)

(8.4)

(IH)

Hence, ($\forall i : i \geq 0 : (\forall j : r_{i-1} < j \leq r_i : [U]_j = r_i)$).  

□

Lemma 21 ($\forall i : i \geq 0 : [T]_i = r_i$).

Proof Proof by induction over $i$.

Base step ($i = 0$).

\[ [T]_0 = [U \text{ fby } U \otimes (T + 1)]_0 \]
\[ = [U]_0 \]
\[ = r_0 \]  

(13.5)

(8.5)

(20)

Induction step ($i = k + 1$).

Suppose ($\forall i : 0 \leq i \leq k : [T]_i = r_i$).

\[ [T]_{k+1} = [U \text{ fby } U \otimes (T + 1)]_{k+1} \]
\[ = [U \otimes (T + 1)]_k \]
\[ = [U]_{[T+1]_k} \]
\[ = [U]_{[T]_k+1} \]
\[ = [U]_{r_k+1} \]
\[ = r_{k+1} \]  

(13.5)

(8.6)

(12)

(8.2)

(IH)

(20)

Hence, ($\forall i : i \geq 0 : [T]_i = r_i$).  

□

Proposition 22 $X \text{ wfr } Y = X \text{ wfr } Y$. 

Proof

\[ [X \text{ wvr } Y]_i = [X @ T]_i \] 

\[ = [X][T]_i \] \hfill (13.4)

\[ = [X]_{r_i} \] \hfill (21)

\[ = [X^{r_i}]_0 \] \hfill (8.2)

\[ = [X^{r_i} \text{ fby } X^{r_i+1} \text{ wvr } Y^{r_i+1}]_0 \] \hfill (8.6)

\[ = [\text{if } [Y]_{r_i} \text{ then } X^{r_i} \text{ fby } X^{r_i+1} \text{ wvr } Y^{r_i+1}] \]

\[ \text{else } X^{r_i+1} \text{ wvr } Y^{r_i+1}]_0 \] \hfill (8.16)

\[ = [\text{if } \text{first } Y^{r_i} \text{ then } X^{r_i} \text{ fby } X^{r_i+1} \text{ wvr } Y^{r_i+1}] \]

\[ \text{else } X^{r_i+1} \text{ wvr } Y^{r_i+1}]_0 \] \hfill (8.12)

\[ = [X^{r_i} \text{ wvr } Y^{r_i}]_0 \] \hfill (9.1)

\[ = [(X \text{ wvr } Y)]_0 \] \hfill (19)

\[ = [X \text{ wvr } Y]_i \] \hfill (8.11)

Hence \((\forall i : i \geq 0 : [X \text{ wvr } Y]_i = [X \text{ wvr } Y]_i)\). Hence \(X \text{ wvr } Y = X \text{ wvr } Y\). \(\square\)

The proofs are now complete.

5 Conclusion

We have defined a broad class of tail-recursive functions that can be transformed into non-recursive functions that use the \text{wvr} and \text{skip} operators. We then showed that the \text{wvr} operator can be itself defined in terms of the \# and @ operators. As a corollary, these tail-recursive functions can all be defined in terms of the \# and @ operators.

It should be clear that the translation given in Section 3 for the tail-recursive functions yields functions that themselves can be readily translated into a reactive program written in an imperative language. We therefore consider the results of this paper to be useful for the compilation of Lucid programs to other languages.

References