Indexical translation of tail-recursive functions

Jean-Raymond Gagné    Joey Paquet    John Plaice

May 19, 1999

Abstract

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 wvr and 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, 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 Bill Wadge on higher-order functions 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, such as the well known wvr and upon functions. Typically, these functions are tail-recursive, of the general form:

\[ F(\bar{c}, \bar{X}) = \beta_0(\bar{c}, \bar{X}) \ fby \ldots \ fby \beta_{06}(\bar{c}, \bar{X}) \ fby \]

\[ \text{if } \alpha_1(\bar{c}, \bar{X}) \text{ then } \beta_{11}(\bar{c}, \bar{X}) \ fby \ldots \ fby \beta_{16}(\bar{c}, \bar{X}) \ fby \ F(\bar{c}_1, \bar{X}_1) \]

\[ \text{else if } \alpha_2(\bar{c}, \bar{X}) \text{ then } \beta_{21}(\bar{c}, \bar{X}) \ fby \ldots \ fby \beta_{26}(\bar{c}, \bar{X}) \ fby \ F(\bar{c}_2, \bar{X}_2) \]

\[ \ldots \]

\[ \text{else } \beta_{p1}(\bar{c}, \bar{X}) \ fby \ldots \ fby \beta_{p6}(\bar{c}, \bar{X}) \ fby \ F(\bar{c}_p, \bar{X}_p) \]

where

*Presented to the International Symposium on Languages for Intensional Programming, Demokritos Institute, Athens, Greece, 28–30 June 1999. Current addresses: Jean-Raymond Gagné, Département d’informatique, Université Laval, Québec (Québec) Canada G1K 7P4. Joey Paquet, Department of Computer Science, Concordia University, Montréal (Québec) Canada H3G 1M8. John Plaice, School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia. Email: gagne@ift.ulaval.ca, paquet@cs.concordia.ca, plaice@cse.unsw.edu.au.*
• The $\bar{c} = (c_1, \ldots, c_m)$ are constants.

• The $\bar{X} = (X_1, \ldots, X_n)$ are input streams.

• The $\alpha_i(\bar{c}, \bar{X})$ are Boolean expressions of the $\bar{c}$ and of specific elements of the $\bar{X}$, i.e. elements of the form $\text{first} (\text{next}^k(X))$, with $k$ fixed.

• The $\beta_{ij}(\bar{c}, \bar{X})$ are expressions of the $\bar{c}$ and of specific elements of the $\bar{X}$.

• The $\bar{c}_i = (c_{i1}, \ldots, c_{im})$ are new constants or specific elements used in the $\alpha_i(\bar{c}, \bar{X})$ or $\beta_{ij}(\bar{c}, \bar{X})$.

• The $\bar{X}_i = (X_{i1}, \ldots, X_{in})$ are permutations of the inputs, possibly advanced using a fixed number of $\text{next}$. Technically, $X_{ij} = \text{next}^{w_i} X_{\theta_i j}$, where $\theta_i$ is a permutation of $(1, \ldots, n)$.

We can also write the above function as

$$F(\bar{c}, \bar{X}) = \beta_0 \; \text{by} \; \prod_{i=1..m} (\alpha_i \rightarrow \beta_i \; \text{by} \; F(\bar{c}_i, \bar{X}_i))$$

where it is understood that the $\alpha_i$ are Boolean expressions and the $\beta_i$ are finite streams of expressions.

2 General direction

In this abstract, we will only give the general structure of the translation. The complete presentation will have to wait for the final paper. However, the general translation can be given in the following terms.

We create four multidimensional dataflows directly from the definition of the program:

• B @.line j @.entry k corresponds to $\beta_{jk}$, with $j = 0..n, \; k = 1..l_j$.

• P @.line j @.var i corresponds to $\theta_{ji}$, the permutation of variable $i$ in line $j$.

• N @.line j @.var i corresponds to $\tau_{ji}$, the advancement of variable $i$ in line $j$.

• A @.line j corresponds to $\alpha_j$, the $j$-th condition.

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 provoke an empty output. This is the case, for example, for the $\text{wvr}$ function. By treating this function separately, then everything can be converted. We give below a formalization of the indexical translation of the $\text{wvr}$ function, first given in the original Lucid book [5].
3 Pipeline dataflows

The origins of Lucid date back to 1974. At that time, Ashcroft and Wadge were working on a purely declarative language in which iterative algorithms could be expressed naturally [1]. Their work fits into the broad area of research into program semantics and verification. It would turn out, serendipitously, that their work would also be relevant to the dataflow networks and coroutines of Kahn and MacQueen [2, 3]. In the original Lucid (whose operators are underlined in the following text), streams were defined in a pipeline manner, with two separate definitions: one for the initial element, and another one for the subsequent elements. For example, the equations

\[
\text{first } X = 0 \\
\text{next } X = X + 1
\]

define variable \(X\) to be a stream, such that

\[
x_0 = 0 \\
x_{i+1} = x_i + 1
\]

In other words,

\[X = (x_0, x_1, \ldots, x_i, \ldots) = (0, 1, \ldots, i, \ldots)\]

Similarly, the equations

\[
\text{first } X = X \\
\text{next } Y = Y + \text{next } X
\]

define variable \(Y\) to be the running sum of \(X\), i.e.

\[
y_0 = x_0 \\
y_{i+1} = y_i + x_{i+1}
\]

In other words,

\[Y = (y_0, y_1, \ldots, y_i, \ldots) = (0, 1, \ldots, \frac{i(i+1)}{2}, \ldots)\]

It soon became clear that a new operator \(\text{fby} (\text{followed by})\) could be used to define the typical situation. Hence the above two variables could be defined as follows:

\[
X = 0 \text{ fby } X + 1 \\
Y = X \text{ fby } Y + \text{next } X
\]

Hence, we can summarize the three basic operators of the original Lucid.
Definition 1

If \( X = (x_0, x_1, \ldots, x_i, \ldots) \) and \( Y = (y_0, y_1, \ldots, y_i, \ldots) \), then

\[
\begin{align*}
(1) \text{ first } X & \overset{\text{def}}{=} (x_0, x_0, \ldots, x_0, \ldots) \\
(2) \text{ next } X & \overset{\text{def}}{=} (x_1, x_2, \ldots, x_i+1, \ldots) \\
(3) X \text{ fby } Y & \overset{\text{def}}{=} (x_0, y_0, y_1, \ldots, y_i-1, \ldots)
\end{align*}
\]

Clearly, analogues can be made to list operations, where \( \text{first} \) corresponds to \( \text{hd} \), \( \text{next} \) corresponds to \( \text{tl} \), and \( \text{fby} \) corresponds to \( \text{cons} \).

When these operators are combined with Landin’s ISWIM [4] (If You See What I Mean), essentially typed \( \lambda \)-calculus with syntactic sugar, it becomes possible to define complete Lucid programs. The following three derived operators have turned out to be very useful (we will use them later in the text):

Definition 2

\[
\begin{align*}
(1) X \text{ wyr } Y & \overset{\text{def}}{=} \text{if } \text{first } Y \text{ then } X \text{ fby } (\text{next } X \text{ wyr } \text{ next } Y) \\
& \quad \text{else } (\text{next } X \text{ wyr } \text{ next } Y) \\
(2) X \text{ asa } Y & \overset{\text{def}}{=} \text{first } (X \text{ wyr } Y) \\
(3) X \text{ upon } Y & \overset{\text{def}}{=} X \text{ fby } (\text{if } \text{first } Y \text{ then } (\text{next } X \text{ upon } \text{ next } Y) \\
& \quad \text{else } (X \text{ upon } \text{ next } Y))
\end{align*}
\]

Where \( \text{wyr} \) stands for \textit{whenever}, \( \text{asa} \) stands for \textit{as soon as} and \( \text{upon} \) stands for \textit{advances upon}.

4 Tagged-token dataflows

With the original Lucid operators, one could only define programs with pipeline dataflows, i.e. in which the \((i+1)\)-th element in a stream is only computed once the \(i\)-th element has been computed. This situation is potentially wasteful of resources, since the \(i\)-th element might not necessarily be required. More important, it only allows sequential access into streams.

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). We have set out on the road to intensional programming. Everything is now redefined in terms of the operators \# and @. In terms of the three original operators, these are defined as:
Definition 3

(1) \# \quad \text{def} \quad 0 \text{fby} \ (\# + 1)
(2) X \@ Y \quad \text{def} \quad \text{if } Y = 0 \text{ then } \text{first} \ X \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else} \ (\text{next} \ X) \@ (Y - 1)

Below, we will give definitions for the original operators using these two new operators. In so doing, we will use the following axioms.

Axiom 1 \textbf{Let } i \geq 0.

\begin{align*}
(1) \ [c]_i &= c \\
(2) \ [X + c]_i &= [X]_i + c \\
(3) \ [\text{first} \ X]_i &= [X]_0 \\
(4) \ [\text{next} \ X]_i &= [X]_{i+1} \\
(5) \ [X \text{ fby } Y]_0 &= [X]_0 \\
(6) \ [X \text{ fby } Y]_{i+1} &= [Y]_i \\
(7) \ \text{if true then } [X]_i \text{ else } [Y]_i &= [X]_i \\
(8) \ \text{if false then } [X]_i \text{ else } [Y]_i &= [Y]_i \\
(9) \ \text{if } C \text{ then } X \text{ else } Y]_i &= \text{if } C]_i \text{ then } [X]_i \text{ else } [Y]_i
\end{align*}

Before we give the new definitions of the standard Lucid operators, we show some basic properties of \@ and \#. We will use throughout the discussion here \([X]_i\) instead of \(x_i\), as it allows for greater readability. Furthermore, we will, as is standard, write \(X = Y\) whenever we have

\((\forall i : i \geq 0 : [X]_i = [Y]_i)\)

Proposition 1 \textbf{Let } i \geq 0.

\begin{align*}
(1) \ [\#]_i &= i \\
(2) \ [X \@ Y]_i &= [X]_y_i
\end{align*}
Proof

(1) Proof by induction over $i$.
   Base step ($i = 0$).

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

   Induction step ($i = k + 1$). Suppose $(\forall i : i \leq k : [#]_i = i)$.

   $[#]_{k+1} = \text{[by $(# + 1)$]}_{k+1}$
   $= [# + 1]_k$
   $= [#]_k + 1$
   $= k + 1$

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

(2) Let $i \geq 0$. We will prove by induction over $y_i$ that $y_i \geq 0 \Rightarrow [X \ominus Y]_i = [X]_{[y]_i}$.
   Base step ($y_i = 0$).

   $[X \ominus Y]_i = \text{[if $Y = 0$ then $\text{first} X$ else $(\text{next} X) \ominus (Y - 1)$]}_i$
   $= \text{if $[Y = 0]_i$ then $\text{first} X_i$ else $[(\text{next} X) \ominus (Y - 1)]_i$}$
   $= \text{if $[Y]_i = 0$ then $\text{first} X_i$ else $[(\text{next} X) \ominus (Y - 1)]_i$}$
   $= \text{first} X_i$
   $= \text{[X]_0}$
   $= \text{[X]_[y]_i}$

   Induction step ($y_i = k + 1$). Suppose $(\forall i : i \leq k : [#]_i = i)$.

   $[X \ominus Y]_i = \text{[if $Y = 0$ then $\text{first} X$ else $(\text{next} X) \ominus (Y - 1)$]}_i$
   $= \text{if $[Y = 0]_i$ then $\text{first} X_i$ else $[(\text{next} X) \ominus (Y - 1)]_i$}$
   $= \text{if $[Y]_i = 0$ then $\text{first} X_i$ else $[(\text{next} X) \ominus (Y - 1)]_i$}$
   $= [(\text{next} X) \ominus (Y - 1)]_i$
   $= \text{[next} X]_{[y]_{i-1}}$
   $= \text{next} X]_{[y]_{i-1}}$
   $= \text{[X]_{[y]_{i-1+1}}}$
   $= \text{[X]_{[y]_i}}$

   Hence $(\forall i : i \geq 0 : [Y]_i \geq 0 \Rightarrow ([X \ominus Y]_i = [X]_{[y]_i})$. □
Definition 4

(1) \( \text{first } X \) \( \overset{\text{def}}{=} X @ 0 \)

(2) \( \text{next } X \) \( \overset{\text{def}}{=} X @ (# + 1) \)

(3) \( X \text{ fby } Y \) \( \overset{\text{def}}{=} \text{if } # = 0 \text{ then } X \text{ else } Y @ (# - 1) \)

(4) \( X \text{ wvr } Y \) \( \overset{\text{def}}{=} X @ T \)
   
   where
   
   \( T = U \text{ fby } U @ (T + 1) \)
   
   \( U = \text{if } Y \text{ then } # \text{ else } \text{next } U \)
   
   end

(5) \( X \text{ asa } Y \) \( \overset{\text{def}}{=} \text{first } (X \text{ wvr } Y) \)

(6) \( X \text{ upon } Y \) \( \overset{\text{def}}{=} X @ W \)
   
   where
   
   \( W = 0 \text{ fby } Y \text{ if } Y \text{ then } (W + 1) \text{ else } W \)
   
   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 2 \( \text{first } X = \text{first } X. \)

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

\[
[\text{first } X]_i = [X @ 0]_i \\
= [X]_{[0]_i} \\
= [X]_0 \\
= [\text{first } X]_i
\]

Hence \( \text{first } X = \text{first } X. \)

Proposition 3 \( \text{next } X = \text{next } X. \)

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

\[
[next X]_i = [X @ (# + 1)]_i \\
= [X]_{[#+1]_i} \\
= [X]_{[#]_i+1} \\
= [X]_{i+1} \\
= [\text{next } X]_i
\]

Hence \( \text{next } X = \text{next } X. \)
Proposition 4 \( X \text{ fby } Y = X \text{ fby } Y \).

Proof  Proof by induction over \( i \).

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

\[
[X \text{ fby } Y]_0 = \begin{cases} 
\text{if } \# = 0 \text{ then } X \text{ else } Y \oplus (# - 1)_0 \\
\end{cases} 
\]

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

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

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

The proof for \( \forall x \) 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 the text in Definitions 4.4.1 and 4.4.2. In addition, we must define the rank of a Boolean stream. Finally, we will have to introduce another set of axioms, that allow us to compare two entire streams, as opposed to particular elements in the two streams.

Axiom 2 Let \( i \geq 0 \).

\[
\begin{align*}
(1) & \quad X^0 = X \\
(2) & \quad [X^i] = [X]_i \\
(3) & \quad \text{first } X^i = [X]_i \\
(4) & \quad \text{next } X^i = X^{i+1} \\
(5) & \quad \text{next } (X \text{ fby } Y) = Y \\
(6) & \quad \text{first } X \text{ fby } Y = X \text{ fby } Y \\
(7) & \quad \text{if true then } X \text{ else } Y = X \\
(8) & \quad \text{if false then } X \text{ else } Y = Y
\end{align*}
\]
Definition 5  Let $Y$ be a Boolean stream.

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

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

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

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

Proof  Let $i \geq -1$. Proof by downwards induction over $j$. Note that $r_i < r_{i+1}$.

Base step ($j = r_{i+1}$).

\[X^{r_{i+1}} \text{ wfr } Y^{r_{i+1}} = X^{r_{i+1}} \text{ wfr } Y^{r_{i+1}}\]  
Identity

Induction step ($j = k - 1, j > r_i$).

\[X^{k-1} \text{ wfr } Y^{k-1} = \begin{cases} \text{if } \text{first } Y^{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wfr } Y^k \\ \text{else } X^k \text{ wfr } Y^k \end{cases}\]  
Defn. 2.1

\[= \begin{cases} \text{if } [Y]_{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wfr } Y^k \\ \text{else } X^k \text{ wfr } Y^k \end{cases}\]  
Axiom 2.3

\[= X^k \text{ wfr } Y^k\]  
Axiom 2.8

\[= X^{r_{i+1}} \text{ wfr } Y^{r_{i+1}}\]  
Ind. Hyp.

Hence, $(\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : X^j \text{ wfr } Y^j = X^{r_{i+1}} \text{ wfr } Y^{r_{i+1}}))$.  \[\square\]

Lemma 2  $(\forall i : i \geq 0 : (X \text{ wfr } Y)^i = X^{r_i} \text{ wfr } Y^{r_i})$.

Proof  Proof by induction over $i$.

Base step ($i = 0$).

\[(X \text{ wfr } Y)^0 = X \text{ wfr } Y = X^0 \text{ wfr } Y^0 = X^{r_0} \text{ wfr } Y^{r_0}\]  
Axiom 2.1

Induction step ($i = k + 1$).
\[(X \text{ wfr } Y)^{k+1} = \text{ next } ((X \text{ wfr } Y)^k) = \text{ next } (X^{r_k} \text{ wfr } Y^{r_k}) = \text{ next } (\text{ if first } Y^{r_k} \text{ then } X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1} \text{ else } X^{r_k+1} \text{ wfr } Y^{r_k+1}) = \text{ next } (\text{ if } [Y]_{r_k} \text{ then } X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1} \text{ else } X^{r_k+1} \text{ wfr } Y^{r_k+1}) = \text{ next } (X^{r_k} \text{ fby } X^{r_k+1} \text{ wfr } Y^{r_k+1}) = X^{r_k+1} \text{ wfr } Y^{r_k+1} = X^{r_k+1} \text{ wfr } Y^{r_k+1}\]

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

**Lemma 3** \((\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : [U]_j = r_{i+1})\).

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

**Base step** \((j = r_{i+1})\).

\([U]_{r_{i+1}} = [\text{ if } Y \text{ then } \# \text{ else } \text{ next } U]_{r_{i+1}} = [Y]_{r_{i+1}} \text{ then } [\#]_{r_{i+1}} \text{ else } \text{ next } U]_{r_{i+1}} = [\#]_{r_{i+1}} = r_{i+1}\)

**Induction step** \((j = k - 1, j > r_i)\).

\([U]_{k-1} = [\text{ if } Y \text{ then } \# \text{ else } \text{ next } U]_{k-1} = [Y]_{k-1} \text{ then } [\#]_{k-1} \text{ else } \text{ next } U]_{k-1} = \text{ next } U]_{k-1} = [U]_k = r_{i+1}\)

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

**Lemma 4** \((\forall i : i \geq 0 : [T]_i = r_i)\).

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

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

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

**Defn. 4.4.1**

\[\text{Axiom 1.5}\]

\[\text{Lemma 3}\]
Induction step $i = k + 1$.

$[T]_{k+1} = [U \circby U @ (T + 1)]_{k+1}$
$= [U @ (T + 1)]_k$
$= [U]_{T+1,k}$
$= [U]_{T+1,k+1}$
$= r_{k+1}$

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

**Proposition 5** $X \text{wvr} Y = X \text{wvr} Y$.

Proof

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