Declarative Multithreaded Programming

Blanca Mancilla     John Plaice
School of Computer Science and Engineering
University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{mancilla, plaice}@cse.unsw.edu.au
2009

Abstract

We demonstrate how TransLucid can be used as a reactive system by introducing sets and privileged dimensions for time and port in the language. At each instant, there is a set of active ports, where sets of equations, demands and threads are all registered. Each thread defines a sequence of (state, demand) pairs, and threads may interact through the overall set of equations. The entire system remains fully declarative.

Keywords: Synchronous programming, distributed computing, declarative programming, Cartesian programming, multidimensional programming.

1 Introduction

In this paper, we apply the principles of synchronous programming to the extension of the TransLucid language in order to study many aspects of reactive, distributed and mobile computing. The intuitive idea is that certain dimensions have a natural physical interpretation, and that the current set of valid equations varies according to these dimensions. In this manner, “change through time and space” (real dimensions) can be modeled as variation through an index space (which may include both real and virtual dimensions), as is consistent with Cartesian programming.

The synchronous TransLucid system presented in this paper is an extension of the simple TransLucid system [3], which consists of a set of equations and a demand for the evaluation of an expression in a given context. In the synchronous system, the set of equations grows over time. At each instant t, there is a set of open ports \( p \in P_t \). For each pair \( (p, t) \), new equations may be added to the system, and new demands may be made as well.

Introducing dimension time into TransLucid allows its use for the programming of reactive systems, and TransLucid can then be regarded as a super-LUSTRE, a generalised LUSTRE [2] where the time dimension is part of the index set as opposed to the only index. Introducing the port dimension creates the potential for distributed and mobile computing, as the port defines a physical visible position in a physical grid. However this is not sufficient as time is crucial for mobility: Is port A at position \( (x, y, z) \) available for me to land now? To incorporate time we need to define, unorthodoxically, the thread. A thread is a sequence of \( (state_t, demand_t) \) pairs where, at each instant \( t \), \( state_t \) defines the current context for the thread and \( demand_t \) defines the thread’s current action. We can envisage a mobile device moving at each instant from one place to another and then executing some action.

These views of port and thread may seem far away from the standard networking view of port and the computational view of thread. However, at a sufficiently abstract level, our view is in fact consistent with the traditional views.
Since a TransLucid system is entirely demand-driven, communication can only take place between threads if they can be made aware of one another. To do this requires a centralised mechanism that can keep track of all of the active ports, threads, equations and demands, which in turn requires adding set-based primitives to TransLucid. In fact, this paper could be entitled, “Adding sets to TransLucid”! There are sets of instants, sets of ports, sets of sets of equations, sets of demands, and so on.

The order of presentation will be bottom-up. We begin by a brief presentation of TransLucid (§2), followed by adding sets to TransLucid expressions (§3–§4). We then show (§5) how best-fitting can be used to choose the most appropriate definition for an identifier when there are several possibilities. Once the preliminaries are finished, we present (§6) the overall structure of a synchronous TransLucid system and its semantics. We then show (§7) that the synchronous ideas can be extended to distributed systems where the different ports do not have identical clocks, without losing the declarative nature of TransLucid. We conclude (§8) with a discussion of future work.

2 Background

TransLucid [3, 4] is a declarative language in which expressions vary in a multidimensional context, where any ground value may play the rôle of dimension. During the evaluation of an expression, the context may be queried, dimension by dimension, in order to adapt its behaviour to the context. During evaluation of an expression, the context may be changed as well.

Below, we give a simple example defining the Fibonacci, factorial and Ackerman functions.

```
infixn "==" "operator==" 1;;
infixl "+" "operator+" 2;;
infixl "-" "operator-" 2;;
infixl "+" "operator*" 3;;

fib = if #0 _==_ 0 then 1
elseif #0 _==_ 1 then 1
  else fib@[0:#0-2] + fib@[0:#0-1]
fi;;

fact = if #0 _==_ 0 then 1
  else #0 * fact@[0:#0-1]
fi;;

ack = if #0 _==_ 0 then #1+1
  elseif #1 _==_ 0
    then ack@[0:#0-1,1:1]
    else ack@[0:#0-1,1:ack@[1:#1-1]]
  fi;;

fib@[0:4];;
fact@[0:4];;
ack@[0:3,1:4];;
```

TransLucid is currently used as a coördination language above C++. The first four lines inform the parser of the associativity and precedence of the infix operators appearing in the program, and also map the operator names to the corresponding C++ functions.

In the expressions, the two key TransLucid operations are #, for querying the current context, and @, for modifying the current context.

Below is the returned result:

```
5 ;;
24 ;;
125 ;;
```
Every aspect of TransLucid is sensitive to the context. Even the parsing and printing of constants is context-dependent. Below, we show some of the possibilities that can be done with the parsing and printing of large numbers, of Système International (SI) units, and of French verbs.

library "si";;
library "verb";;

%%
1000000000000000000000000000000000 @
   [outintmp:true,
    outtext:true,
    outenames:true] ;;
si<Hz> @
   [instyle:"symbol",
    outstyle:"name"];;
si<newton> @
   [instyle:"name",
    outstyle:"basicunits"];;
si<electric capacitance> @
   [instyle:"quantity",
    outstyle:"name"];;
verb<envoyer> @
   [outmode:"indicative",
    outtense:"future",
    outperson:"1s"];;
verb<assiègeai> @
   [inmode:"indicative",
    intense:"past", inperson:"1s"];;

In this example, the si library defines the si atomic data type, along with dimensions instyle and outstyle, while the verb library defines the verb data type, along with dimensions for mode, tense and person, both input and output.

Below is the returned result:

intmp<one trilliard> ;;
si<hertz> ;;
si<m.kg.s-2> ;;
si<farad> ;;
verb<enverrai> ;;
verb<assiège> ;;

An initial set of rules was provided in [1] and the language's initial goal was to be usable as a real programming language, directly or as a target for other paradigms. Intermediate implementations were developed, but the multithreaded implementation presented in [5], let to the introduction of lazy tuples in which the codomain of a context is evaluated lazily. TransLucid is now being implemented as a production language, allowing for user-defined data types and operations.

The current version of TransLucid is described in detail in [3], which covers the syntax, denotational semantics and operational semantics of the language. The evaluation of a TransLucid expression uses a warehouse to cache previously computed (identifier, context) pairs, ensuring that only those dimensions referred to in the computation of the corresponding expressions are used for caching purposes.

In the current paper, we present an approach to extending TransLucid to incorporate two dimensions with clear physical interpretation: port and time. These two dimensions may be used as virtual dimensions, but they are also used to keep track of sets of equations, demands and threads to the system. However, sets first need to be added to TransLucid.
3 Sets as values

In this section, we add sets to TransLucid, by introducing a new built-in type, called \texttt{set}, that plays a dual role. At one level, \texttt{set} and the standard operations of set enumeration, comprehension, union, intersection and difference, can be viewed as an abstract data type. At another level, a set can be considered to be a container of objects, and one might wish to iterate over these objects and to do some treatment thereon. Rather than providing a special set of iterators over such structures, we provide means for translating a set into an entity varying in a specified dimension.

The \textit{extensional} view of the set provides the standard set of operations:

\[\{E,\ldots,E\}\] creation of set by enumeration of elements;
\[\{E \mid E\}\] set comprehension;
\[S \cup S\] set union;
\[S \cap S\] set intersection;
\[S - S\] set difference.

The \textit{intensional} view of the set allows one to access individual elements by changing the value associated with a specified dimension. There are two operators:

\[h = \text{settoHD}.d(S)\]: Create a hyperdaton (a TransLucid object) varying in dimension \(d\). If \(S = \{c_0,\ldots,c_{n-1}\}\) has \(n\) elements, then \(h @ [d : i]\) equals \(c_i\) if \(0 \leq i < n\), and \text{sp<eod>} otherwise.

\[S = \text{HDtoiset}.d(h)\]: Create a set from a hyperdaton varying in dimension \(d\). If \(h[d,i]\) is non-eod for all \(i\) such that \(i = 0..n - 1\), and \text{sp<eod>} elsewhere, then \(S\) will be a set with \(n\) elements.

4 Regions

In the following sections, we will define context-dependent equations, equation sets, and demands. To do this, we need to define the \textit{region}, which corresponds to a set of contexts.

\[
\begin{align*}
\text{region} &::= \left[\text{range}\right] \\
\text{range} &::= c \colon c \ldots c \\
&\quad | \quad c : c \ldots \#c \\
&\quad | \quad \#c : c \ldots c
\end{align*}
\]

where the plain \(c\)'s specify constants and the boldfaced \(c\)'s specify dimensions. The multirange specifies a set of contexts, by defining the set of acceptable values for a set of dimensions. The \#\(c\) entries allow the limits of a dimension to be defined in terms of the current value of another dimension (no circularities are permitted), as would be needed, for example, to specify triangular matrices.

In order to use the context-dependent equations, one needs to be able to compare two regions, to determine if one is included in the other. This is done with the \(\subseteq\) operator, defined below.

Let \(R\) be a region. The domain of \(R\) is the set of left-hand sides in \(R\). Suppose \(R = \{c \mapsto r_c \mid c \in \text{dom } R\}\), where each \(r_c\) is of one of the forms:

\[(c_{\text{min}}, c_{\text{max}})\]
\[(c_{\text{min}}, \#c_{\text{max}})\]
\[(\#c_{\text{min}}, c_{\text{max}})\]
Then \( \text{ran}_R(c) \) is defined by:

\[
\text{ran}_R(c) = \begin{cases} 
[c_{\min}, c_{\max}], & r_c = (c_{\min}, c_{\max}) \\
[c_{\min}, \max\{\text{ran}_R(c_{\max})\}], & r_c = (c_{\min}, \#c_{\max}) \\
\min\{\text{ran}_R(c_{\min})\}, c_{\max}], & r_c = (\#c_{\min}, c_{\max})
\end{cases}
\]

Let two regions \( R \) and \( R' \), and suppose \( c \in \text{dom} \, R \) and \( c \in \text{dom} \, R' \) for some \( c \). We wish to compare \( r_c \) and \( r'_{c'} \). First

\[
r_c \sqsubseteq^{\text{min}} r'_{c'} = \begin{cases} 
c \geq c', r_{\min} = c' \text{ and } r'_{\min} = c' \\
c \geq \min\{\text{ran}_R(c_{\min})\}, r_{\min} = c' \text{ and } r'_{\min} = \#c'_{\min} \\
c_{\min} = c'_{\min}, r_{\min} = \#c_{\min} \text{ and } r'_{\min} = \#c'_{\min}
\end{cases}
\]

\[
r_c \sqsubseteq^{\text{max}} r'_{c'} = \begin{cases} 
c \leq c', r_{\max} = c' \text{ and } r'_{\max} = c' \\
c \leq \max\{\text{ran}_R(c_{\max})\}, r_{\max} = c' \text{ and } r'_{\max} = \#c'_{\max} \\
c_{\max} = c'_{\max}, r_{\max} = \#c_{\max} \text{ and } r'_{\max} = \#c'_{\max}
\end{cases}
\]

Now we can compare \( R \) and \( R' \). We have that: \( R \sqsubseteq R' \) iff \( \text{dom} \, R' \subseteq \text{dom} \, R \) and \( \forall c \in \text{dom} \, R' \), \( r_c \sqsubseteq^{\text{min}} r'_{c'} \) and \( r_c \sqsubseteq^{\text{max}} r'_{c'} \).

To determine if a context \( \kappa \) is included in a region \( R \), we use the \( \in \) operator, defined as follows: \( \kappa \in R \) iff \( \{ \kappa \mid \text{dom} \, R \} \subseteq R \).

### 5 Equation sets and best-fit definitions

In the synchronous TransLucid system presented in the next section, the overall set of equations is evolving through time and space. For each identifier \( x \), there may be several defining equations for \( x \). Furthermore, for each instant \( t \), the set of equations defining \( x \) may be different.

An equation set is a set \( Q \) of 4-tuples of the form \( q = (x_q, \kappa_q, K_q, E_q) \), where

- \( x_q \) is the identifier being defined;
- \( \kappa_q \) is the context in which the definition was added to the equation set; currently, we are interested in the value of \( \kappa_q(\text{time}) \), i.e., the timestamp at which \( q \) entered the system;
- \( K_q \) is a set of contexts in which the definition is valid;
- \( E_q \) is the expression.

The concrete syntax for context-dependent equations is as follows:

\[
eqn ::= \text{ident} @ \text{region} = \text{expr}
\]

Suppose we have an equation set \( Q \). Suppose we wish to evaluate identifier \( x \) in context \( \kappa \). We define:
\[ Q_x = \{ q \in Q \mid x_q = x \} \]

\[ Q_{x^\kappa} = \{ q \in Q_x \mid \kappa_q(\text{time}) \leq \kappa(\text{time}) \land \kappa \in K_q \} \]

The best-fit definition is the unique \( \eta \in Q_{x^\kappa} \) such that \( \forall q \in Q_{x^\kappa}, K_q \sqsubseteq K_\eta \), where \( \sqsubseteq \) was defined in §4.

We then write \( Q(x, \kappa) = E_\eta \), if \( \eta \) exists, \text{sp<undef>} otherwise.

6 Synchronous TransLucid system

A synchronous TransLucid system

\[ S = (H, L, P, Q, D, P, Q, D, T) \]

is a 10-tuple, where:

- \( H \) is a header defining the translation from concrete to abstract syntax.
- \( L \) is a set of libraries defining the available types and operations.

\[ P : \text{Time} \to \text{PortID} \]

\[ P : \text{PortID} \to \text{Time} \]

\[ \to \text{EqnID} \times \text{DemandID} \times \text{ThreadID} \]

\[ Q : \text{Time} \to \text{EqnID} \]

\[ Q : \text{EqnID} \to \text{Time} \to \text{PortID} \times \text{Region} \times \text{Eqn}^+ \]

\[ D : \text{Time} \to \text{DemandID} \]

\[ D : \text{DemandID} \to \text{Time} \to \text{PortID} \times \text{Expr}^+ \]

\[ A : \text{Time} \to \text{ThreadID} \]

\[ A : \text{ThreadID} \to \text{Time} \to \text{PortID} \times \text{Eqn}^+ \]

For each port, equation set, demand set and thread, there is a unique identifier \( u \), valid for all of eternity, even if it is decommissioned. This simplifies perfect recall.

**Ports.** (P) There are two operations for manipulating ports.

\[ p = \text{newport} \]

\[ \text{decommissionport} p \]

Let \( t \) be the current instant. Suppose that:

\[ P^\text{new}_t \] is the set of new ports;
\[ P^\text{old}_t \] is the set of decommissioned ports.

Then:

\[ P_t = P_{t-1} \cup P^\text{new}_t - P^\text{old}_t \]

\[ P_{p,t} = (Q_{p,t}, D_{p,t}, A_{p,t}) \]

**Equation sets.** (Q) There are four operations for manipulating equation sets.

\[ q = \text{neweqn} \]

\[ \text{decommissioneqn}(q) \]
moveeqn(q, p)
addtoeqn(q, Q)

Let $t$ be the current instant and $p$ the current port. Suppose that:

- $Q_{p,t}^{\text{new}}$ is the set of new equation sets;
- $Q_{p,t}^{\text{old}}$ is the set of decommissioned equation sets;
- $Q_{p',p,t}^{\text{move}}$ is the set of equation sets that are moving from port $p'$ to port $p$;
- $Q_{p,t}^{\text{add}}$ is a set of pairs of the form $(q, Q)$, meaning the equation set $q$ is being augmented with equation set $Q$.

Then:

$$Q_{p,t} = \bigcup_{p' \in P_{t-1}} Q_{p',p,t}^{\text{move}} \cup Q_{p,t-1} \cup Q_{p,t}^{\text{new}} - Q_{p,t}^{\text{old}}$$

$$Q_{q,t} = Q_{q,t-1} \cup Q, \text{ if } \text{addtoeqn}(q, Q)$$

Demands. (D) There are four operations for manipulating demands.

- $d = \text{newdemand}$
- $\text{decommissiondemand}(d)$
- $\text{movedemand}(d, p)$
- $\text{addtodemand}(d, E)$

Let $t$ be the current instant and $p$ the current port. Suppose that:

- $D_{p,t}^{\text{new}}$ is the set of new demands;
- $D_{p,t}^{\text{old}}$ is the set of decommissioned demands;
- $D_{p',p,t}^{\text{move}}$ is the set of demands that are moving from port $p'$ to port $p$.
- $D_{p,t}^{\text{add}}$ is a set of pairs of the form $(d, E)$, meaning the demand set $d$ is being augmented with demand $E$.

Then:

$$D_{p,t} = \bigcup_{p' \in P_{t-1}} D_{p',p,t}^{\text{move}} \cup D_{p,t-1} \cup D_{p,t}^{\text{new}} - D_{p,t}^{\text{old}}$$

$$D_{d,t} = D_{d,t-1} \cup E, \text{ if } \text{addtodemand}(d, E)$$

Threads. (A) There are four operations for manipulating threads.

- $a = \text{newthread}$
- $\text{decommissionthread}(a)$
- $\text{movethread}(a, p)$
- $\text{addtothread}(a, Q)$

Let $t$ be the current instant and $p$ the current port. Suppose that:

- $A_{p,t}^{\text{new}}$ is the set of new threads;
- $A_{p,t}^{\text{old}}$ is the set of decommissioned threads;
$\mathcal{A}_{p',p,t}^\text{move}$ is the set of threads that are moving from port $p'$ to port $p$.

$\mathcal{A}_{p,t}^\text{add}$ is a set of pairs of the form $(a,Q)$, meaning the thread $a$ is being augmented with equations $Q$ defining variables state and demand.

Then:

$$\mathcal{A}_{p,t} = \bigcup_{p' \in P_{t-1}} \mathcal{A}_{p',p,t}^\text{move} \cup \mathcal{A}_{p,t-1} \cup \mathcal{A}_{p,t}^\text{new} - \mathcal{A}_{p,t}^\text{old}$$

$$\mathcal{A}_{a,t} = \mathcal{A}_{a,t-1} \cup Q, \text{ if } \text{addtodemand}(a,Q)$$

Semantics. The semantics of a system is straightforward:

For each instant $t$
For each port $p \in P_t$
For each demand set $d \in D_{p,t}$
For each demand $E \in d$
Execute $E @ \text{[time : } t\text{]}$
For each thread $a \in \mathcal{A}_{p,t}$
Execute $(\text{demand } \circ \text{state})$
@ $\text{[time : } t, \text{thread : } a\text{]}$

7 Local clocks

In the previous section, we supposed that there was a single clock shared by all ports, and that communication between different threads, even if on different ports, could take place instantaneously. In so doing, we showed that it is possible to refer to threads in a declarative manner. However, as should be clear, the synchronous model does not scale up to more general situations. Nevertheless, it is possible the previous section’s solution can be adapted to allow different ports to have their own clocks and even for variable lag in communication among the ports.

Suppose that every port has its own clock, and that the clocks of the different ports are not synchronised, but all respect the condition of monotonicity, i.e. no clocks run backwards. Then demands can be made using the following dimensions: the port dimension, for each port $p$, the $\text{localtime}_p$ dimension.

When a demand is initiated, it is tagged with the current port $p$ and the current time $t_p$. Should this demand ultimately generate a subdemand to another port $p'$ at time $t_{p'}$, then the evaluator at $p'$ will register the triple $(p,t_p,t_{p'})$. The demand is then handled, and the result is tagged with dimension $\text{localtime}_{p'} = t_{p'}$.

Should another subdemand initiated from the same original demand be made at $p'$, even at a later time, then the time $t_{p'}$ will be used. As a result, any demand originating from the same $(p,t)$ pair will be treated consistently at any given port: equations from different time stamps will not be mixed. Although determinism in the behaviour of a distributed system as described above is not possible, replicability is. Given the return tag of the results of demands, it is possible to create a new demand that will exactly replicate the results of the initial demand.

8 Conclusion

In a synchronous TransLucid system, it is possible to—completely declaratively—support many different styles of programming. A standard single-loop reactive system can be defined using just a single set of equations and a single set of demands. A software system where the software is updated in real time can be defined using multiple sets of equations. A deterministic multithreaded system can be built using multiple threads.

Future work involves developing real applications and extending the model to diverse forms of distributed, mobile and pervasive computing, each involving slightly different forms of demand.
References


