The Cartesian Approach to Context

John Plaice Blanca Mancilla
School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2032, Australia
{plaice,mancilla}@cse.unsw.edu.au
2010

Abstract

We present a new approach for context-oriented programming in which the context is represented by a set of \((\text{dimension},\text{value})\) pairs. This tuple parameterizes the environment, and it can be referred to either as a single entity or as a composed entity, parts of which can independently be accessed. The context is also an index into any programmable entity, in our model the hyperdatons, which are in turn, arbitrary-dimensional arrays of arbitrary extent.

The context may have privileged dimensions and one such dimension is \text{time}, which has as well a physical interpretation. The importance of this dimension relies on the fact that its proper handling will allow the control of software evolution, of systems, and of system instances or views; partial changes or updates to specific parts of a system; and synchronous communications between heterogeneous components or even systems. In fact, it is our tool to create synchronous Cartesian systems, essential for context-aware distributed systems.

The implementation of a Cartesian distributed system may rely on the behavior of several subsystems, all running on an internal clock necessarily infinitely faster than the external one, since a bunch of tasks in a subsystem, corresponds to one tick of the system. These subsystems all run with respect to a shared context called an \text{ether}, which facilitates communication by broadcasting between systems at possibly different levels. The \text{ether} in this case is an active context.

1 Introduction

In this paper, we present a model for context-oriented programming based on the assumption that the context (\text{i.e.} the environment) and its variation can be summarized as a set of \((\text{dimension},\text{value})\) pairs that change through time in a synchronous manner.

Cartesian programming [7] provides the platform to naturally manipulate the context, a multidimensional entity, used as an index into an arbitrary-dimensional space in which the programmable objects are found: variables, methods, functions, databases, etc. Just as the Cartesian coordinate system allowed the algebraisation of geometry, Cartesian programming allows the use of a single formalism in which one can develop an entire software system, with multiple heterogeneous components, in a completely declarative way.

TransLucid, a declarative programming language, is the vehicle in which the concepts of Cartesian programming are being developed. In TransLucid, equations are written similarly to differential equations, \text{i.e.}, only pertinent dimensions appear therein; in a Cartesian program, only relevant dimensions need to be mentioned.

Key to this work is the concept of bestfitting: any entity, at any moment, when requested, has a set of valid definitions. The best one is chosen according to the context. Once this selection has taken place, then its defining expression is evaluated.

Cartesian programming can be understood in two complementary ways: as a new paradigm; and as a tool for making it easier to situate, analyze, implement or compare existing paradigms.
For example, from the Cartesian perspective, a function with \( n \) arguments can be understood as an \( n \)-dimensional table and a call to that function then corresponds to the act of indexing that table; a class can be viewed as a table indexed by its instances and the messages that the latter receive; and a loop can be viewed as a mapping from the control points and the current iteration to the values of variables. This allows, in theory at least, the indexing of any system or any paradigm.

The key concepts presented in what is commonly called context-oriented programming [5, 1, 3] have direct correspondences in Cartesian programming: a layer corresponds to a dimension, the context is a tuple, a set of \((\text{dimension-value})\) pairs; a layer activation is a change of context \((\varnothing)\), where the evaluation of an entity can be requested with a concrete given context; reflective access corresponds to context querying \((\#)\), where the values of dimensions can be directly requested; and multidimensional dispatch corresponds to bestfitting, where there are three privileged dimensions in the context.

However, in Cartesian programs, all dimensions are orthogonal, unless stated otherwise. As a result, bestfitting takes place with respect to all active dimensions, unlike layer activation in many context-oriented programming languages, which is done one layer at a time.

Furthermore, in Cartesian programs, one dimension is privileged: \textbf{time}. The context of an object is time-varying and, as a result, the system changes with respect to the context. We therefore consider Cartesian programs through the lens of synchronous reactive systems. This allows for free the manipulation of instants, backtracking, logs, and heterogeneous systems communication needed for distributed computation, just to name a few.

The work leading to the development of Cartesian programming is summarized in [6, 8]. Broadly, it is the result of the influence of four related threads of research.

The intensional logic of Richard Montague, first presented in 1969 [10], was used to give the semantics of natural language utterances such as “The temperature yesterday”. In the intensional view, this utterance is a function mapping contexts to temperatures, not a single temperature (which is called the extension). This is the intuition behind the context mentioned above.

The Lucid language of William Wadge and Ed Ashcroft was first presented in 1976 [2]. In Lucid, variables define infinite streams which can be interpreted as the complete histories of variables. TransLucid is the latest generalization of Lucid.

The Lustre language (Synchronous Real-Time Lucid) of Nicolas Halbwachs and Paul Caspi was first presented in 1984 [4]. In Lustre, variables are streams whose \(n\)-th entry corresponds to the \(n\)-th tick of a clock. This language is at the heart of the Scade system sold by Estérel Technologies. Author Plaice wrote the first semantics and compiler for Lustre.

Possible worlds versioning [6, 9] defines the variance of a software system or of documents as an intension.

2 TransLucid, the language

TransLucid is the first programming language explicitly designed for Cartesian programming. In TransLucid, all entities are \textit{hyperdatons}, or infinite multidimensional objects in all directions, whose points are accessible randomly using the index.

TransLucid includes three kinds of data structures: (1) atomic values, such as integers, Booleans and characters; (2) hyperdatons, infinite tables of arbitrary dimensionality; and (3) tuples, unary functions mapping atomic values to atomic values, used to index the hyperdatons.

In addition to the primitives allowing the evaluation of expressions (constants, variables, operators, \texttt{if–then–else}), the TransLucid core language only contains three additional primitives: tuple creation, modification of the current context by a tuple, and the demand for a value corresponding to a dimension of the current context. Both the evaluation and the definition of a variable are context-dependent. The language can either be used directly or as intermediate language, \textit{i.e.}, as the target language in the compilation of another language.

The TransLucid infinite data structures are obviously not constructible in a physical computer, hence their evaluation is lazy. If the value of a variable is needed in a given context, a demand
is made for the \((identifier, context)\) pair. This demand then provokes a search for the bestfit definition of the variable with respect to that context. The resulting expression is then evaluated, possibly provoking demands for other identifiers in other contexts.

The dimensionality of a hyperdaton is not fixed. For example, the sequence \(\ldots, 0, 1, 2, \ldots\), varying in dimension \(x\), is considered to vary in all other dimensions, but in a constant manner. Hence, if during a computation, only part of the context is relevant, then only that part will be taken into account. It is possible, then, to have infinite tuples of which a finite part would be used.\(^1\)

The TransLucid operational semantics and interpreter also work in a lazy manner, where a cache is used to memorize repeated intermediate calculations. This is a variant of the memoization sometimes used in functional programming, but adapted to a situation in which there is an arbitrary number of dimensions.

A simple TransLucid program consists of a set of equations that defines a number of variables varying according to an arbitrary set of dimensions. Expressions \((E)\) have a simple grammar:

\[
E ::= c \quad \text{constant} \\
    | \ x \quad \text{identifier} \\
    | \ op(E_1, \ldots, E_n) \quad \text{operation} \\
    | \ if\ E_1 \ \text{then} \ E_2 \ \text{else} \ E_3 \ \text{fi} \quad \text{conditional} \\
    | \ #E \quad \text{context query} \\
    | [E_{11} : E_{12}, \ldots, E_{n1} : E_{n2}] \quad \text{tuple creation} \\
    | \ E_2 \ @ \ E_1 \quad \text{relative context change}
\]

where

- \(c\) is a constant;
- \(x\) is an identifier;
- \(op\) is a pointwise strict operator;
- \(#\): assuming that \(E\) evaluates to a dimension, return the value of that dimension in the current context of evaluation.
- \(@\): evaluate \(E_2\) assuming the context to be \(E_1\). Normally, the right-hand side of a context change will be an index created with the index creation operator.

Equations \((q)\) in TransLucid are contextualized:

\[
qu ::= x \ | \ E \ & \ E = E\]

The expression following the \(|\) is an index created by square brackets, and the expression following the \(\&\) is a Boolean expression. Together, they form a guard. The guard is used to state when this equation is valid, or applicable. When an identifier appears in an expression and is to be evaluated, then the bestfit definition for that identifier is chosen.

As example, consider the Ackermann function, varying in dimensions \(n\) and \(m\).

\[
\begin{align*}
\text{ack} \ | \ [n:0, m:0] &= 1 \\
\text{ack} \ | \ [n:0] &= \ #m + 1 \\
\text{ack} \ | \ [m:0] &= \ text{ack} \ @ \ [n : \ #n - 1, m : 1] \\
\text{ack} &= \ text{ack} \ @ \ [n : \ #n - 1, \\
& \hspace{1cm} m : \ text{ack} \ @ \ [m : \ #m - 1]]
\end{align*}
\]

\(^1\)This is our implementation of the layer activation.
It can be viewed as a 2-dimensional table:

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>

The inner recursive call to \(\text{ack}\) only requires changing the value of dimension \(m\), since dimension \(n\) does not change; this is called relative context change.

The concepts of index creation, context change, context query and bestfitting are the key to TransLucid. Below we define these formally.

The objective is to evaluate a single expression \(E\) with respect to a set of equations \(\sigma\) in a single context \(\kappa\), and to produce a single value \(v\), written \(v = [E]_{\sigma\kappa}\), where

- An atomic value \(v \in V\) is written as \(\text{type}(value)\). If this is the case, then \(\tau(v) = \text{type}\) and \(\psi(v) = value\). There are three predefined types used here: \text{bool} for the Boolean values \text{true} and \text{false}, \text{sp} for special erroneous values (akin to exceptions), and \text{tuple}.

- A tuple value is written \(\kappa \in K = (V \rightarrow V)\)

The rules for \(v = [E]_{\sigma\kappa}\) are built up through the structure of the expression \(E\) and are straightforward.

- \(x\):
  \[
  [x]_{\sigma\kappa} = \begin{cases} 
  [\text{best}(x)]_{\sigma\kappa}, & x \in \text{dom} \sigma \\
  \text{sp}(\text{undef}), & \text{otherwise}
  \end{cases}
  \]

- \(c\):
  \[
  [c]_{\sigma\kappa} = c
  \]

- \(\text{op}(E_1, \ldots, E_n)\):
  \[
  \left[\text{op}(E_1, \ldots, E_n)\right]_{\sigma\kappa} = \text{let } v_i = [E_i]_{\sigma\kappa} \text{ in}
  \begin{cases}
  \min \{ v_i \mid \tau(v_i) = \text{sp} \}, & \exists i. \tau(v_i) = \text{sp} \\
  \text{op}(v_1, \ldots, v_n), & \text{otherwise}
  \end{cases}
  \]

- \(\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \text{ fi}\):
  \[
  \left[\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \text{ fi}\right]_{\sigma\kappa} = \text{let } v_1 = [E_1]_{\sigma\kappa} \text{ in}
  \begin{cases}
  v_1, & \tau(v_1) = \text{sp} \\
  \text{sp}(\text{type}), & \tau(v_1) \neq \text{bool} \\
  [E_2]_{\sigma\kappa}, & \psi(v_1) = \text{true} \\
  [E_3]_{\sigma\kappa}, & \psi(v_1) = \text{false}
  \end{cases}
  \]
3 TransLucid Implemented

The TransLucid programming environment (translucid.sourceforge.net) implements the TransLucid language. It is written in C++0x, the new standard for C++. For compilation, it currently requires GNU g++ 4.5.0 (gcc.gnu.org), and the Boost 1.42.0 C++ libraries (www.boost.org). In addition to the three atomic data types described in the previous section, the environment supports mpint (GNU multi-precision integer), uchar (32-bit Unicode character) and ustring (32-bit Unicode string).

At the core of the implementation is the hyperdaton, an arbitrary-dimensional unbounded array. The C++ class HD is given below:

```cpp
class HD
{
  virtual HD() {}
  virtual TaggedValue operator()(const Tuple& k);
};
```

In C++, an object \( x \) instantiating a class \( C \) with the `operator()` method defined is called a functor, or function object: when one writes \( x(\text{actuals}) \), if `operator()`(formals) is defined for \( C \) and of the right type, then it is called; should there be more than one `operator()`(formals) defined for \( C \), i.e., `operator()` is overloaded, then the one with the most specific type for the actuals is called.

The `operator()` method is used to do a lookup in the current context, which is encoded using the data structure Tuple. This lookup might simply require, internally, the return of a constant or a table lookup. Or, it might require some massive computation. The returned result, of type TaggedValue, contains the type, the value and the pertinent subcontext (of the original context passed in as input) that was actually needed to compute the result.

The interpreter consists therefore of a number of classes defining different kinds of hyperdaton, one for each of the different kinds of expression, along with a parser that converts expressions and equations into hyperdatons and a pretty-printer that allows printing of tagged values and
hyperdatons. As a result, it is possible to do Cartesian programming either in C++ or in TransLucid, or through a judicious mix of the two.

The VariableHD hyperdaton is given below:

```cpp
class VariableHD: public HD {
    uuid addExpr(const Tuple& k, HD* h);
};
```

The `addExpr` method is used to add a new definition, itself encoded as a hyperdaton, to the variable hyperdaton. A single variable may have multiple definitions, and when a lookup takes place, the most specific, with respect to the current context, will be used. The `uuid` returned as result from `addExpr` corresponds to a unique identifier generated internally so that the definition may be referred to explicitly.

The VariableHD hyperdaton can also hold a number of variables lower down, so that one can write dot-separated expressions. For example, `x.y.z` means the `z` variable in the `y` variable in the `x` variable.

The SystemHD hyperdaton is derived from VariableHD:

```cpp
class SystemHD: public VariableHD {
    void addInput (const std::map<u32string, HD*>);
    void addOutput(const std::map<u32string, HD*>);
    void addDemand(const u32string& id, const EquationGuard& guard);
    void eval();
};
```

It includes equations and is passed physical hyperdatons as input and output. In this way, it is possible to have hyperdatons that have their own C++-specific interface for use outside of TransLucid, as well as the hyperdaton interface for use within TransLucid. The demands are requests for output hyperdatons to be calculated under certain contexts.

Once all of the equations and demands are added, then the `eval()` method can be called, producing all of the results, and the demands are executed, with the results being placed in the output hyperdatons.

In some sense, everything is a hyperdaton. This approach has simplified the programming of the interpreter. In particular, identifiers and operators are encoded using hyperdatons, through the use of special identifiers.

When one writes `x` as an identifier, the TransLucid interpreter understands this as `ID@[name : x]`. Therefore, all variables are understood to be members of the same hyperdaton, namely `ID`.

Similarly, when one writes `3 + 5`, the TransLucid interpreter will look up the function name, in this case `operator+`, and encode the whole thing as

```cpp
OP @ [name:"operator+", arg0:3, arg1:5]
```

Therefore, all operators are understood to be members of the same hyperdaton, namely `OP`. This approach is used for loading of libraries: when a library is loaded, it is passed the current system; the library then adds to that system equations of the following form:

```cpp
OP @ [name:"operator+",
      arg0:type<intmp>,
      arg1:type<intmp>]
      = <function pointer>
```

As a result, when `operator+` is to be evaluated, the bestfitting process is used to determine which instance of `operator+` is to be used. A similar process is used to keep track of types, dimensions and demands. In time, even the parser and the pretty-printer will be defined this way.
4 Synchronous TransLucid

4.1 Time-varying data

By introducing time into the context as a privileged dimension, the semantics of a Cartesian system $S$ is given a synchronous temporal semantics: at instant $t$, there are instances $S_t$ of the system and $I_t$ of the inputs, and the output $O_t$ of the system is provided by applying $S_t$ to $I_t$. A synchronous reactive system consists of a set of equations and, at each instant $t$, a set of inputs $I_t$ needed to produce a set of outputs $O_t$. Given a TransLucid system described $S(I, O)$, this is done taking into account that each instant $t$ consists of three stages:

- Update the input hyperdatons from the values $I_t$.
- Call $S$.eval().
- Read from the output hyperdatons to retrieve $O_t$.

In this scenario, the set of equations remains fixed, but the input changes with each instant, and presumably so does the output. Only data is time-varying.

If the clock for the current instant is required, then we can use the time dimension, and #time will give its current value. Although this dimension has a physical interpretation, it is still used as an ordinary dimension, except for the fact that one cannot change to a future time (or the past) using the @ operator.

4.2 Time-varying equations

TransLucid actually allows the set of dimensions to vary over time. The interface for VariableHD is actually:

```c++
class VariableHD: public HD
{
    uuid addExpr (const Tuple& k, HD* h);
    void delExpr (const uuid);
    void replExpr(const uuid, const Tuple& k, HD* h);
};
```

and the first stage of the TransLucid reactive system is changed to:

- Update the input hyperdatons from the values $I_t$ and update the system of equations.

We can now properly define the timed behavior of a TransLucid system. Here, we will use three special dimensions whose values are ordered: time (current time), deftime (definition time for equation) and priority (priority level of equation). of a system.

All of the demands for a system are stored in a Boolean hyperdaton called DEMAND. At any instant $t$, the current set of equations is written $Q_t$ and the current set of demands, i.e., the current set of contexts for which DEMAND is true, is written $D_t$.

During the computation phase of an instant, the system goes through the set of demands $D_t$, and evaluates each of them. The output hyperdatons of the system are thus loaded up, and if they are subclassed to have some C++ container interface, then this latter can be used in the C++ code that called the system.
When an identifier $x$ appears in an expression that is evaluated at context $\kappa$ and instant $t$, then the bestfit definition of $x$ must be chosen for $\kappa$ and $t$. The process works as follows:

\begin{verbatim}
for $p = \text{priorityMax}$ downto 0 do
   let $defs = \text{applicableDefs}(x, p, t)$
   if $defs == \emptyset$, next iteration
   let $mdefs = \text{maximalDefs}(defs)$
   let $lmdefs = \text{latestDefs}(mdefs)$
   if $\text{card}(lmdefs) == 1$, return $lmdefs$
   else return $\text{sp}(\text{multidef})$
return $\text{sp}(\text{undef})$
\end{verbatim}

The term $\text{applicableDefs}(x, p, t)$ consists of all those definitions of $x$ for which the guards are valid for priority $p$ and time $t$.

The term $\text{maximalDefs}(defs)$ consists of all the definitions in $defs$ whose context component of the guard is not fully contained in the context component of the guard of any of the other elements of $defs$.

The term $\text{latestDefs}(mdefs)$ consists of all the definitions in $mdefs$ with the maximal $\text{deftime}$ in $mdefs$.

5 Distributed Systems

5.1 Systems within systems

If class $\text{SystemHD}$ is a subclass of class $\text{VariableHD}$, and $\text{VariableHD}$ can hold several other variables below it, then this implies that a system can contain other systems. Given the timed semantics, how does this work? The solution lies in that each level of system nesting implies an infinitely faster clock. If the outer level clock advances with ticks of 1, then the next level in advances with ticks of $\epsilon$ (an infinitesimal value), then the next level in at $\epsilon^2$, and so on.

Each system has its own physical inputs and outputs, defined externally. As a result, we now have a model where multiple levels of system can be defined, with the inner levels running with faster clock rates.

Choosing infinitely faster clocks has the same simplicity as does the original synchronous model. Should one choose a finite amount, the immediate next question would be, “How much?”, and the answer would change from one physical implementation to the next. By stating that the clock is infinitely faster, we need not worry about such problems.

However, in a given level-$\ell$ macro-instant, we cannot allow an infinite number of level-$(\ell + 1)$ micro-instants to take place, as this would mean an infinite amount of work would have to take place in a finite amount of physical time; this clearly is not possible. Therefore, some upper bound would have to be placed on the number of micro-instants for each level. Demands not completed within this number of micro-instants would, depending on the requests, either timeout or else be continued in subsequent macro-instants.

5.2 The æther: sharing within a system

The description has supposed that the evaluation of expressions is context-dependent. But how does one make all of the behavior within an entire system context-dependent? By introducing another special variable, $\text{ÆTHER}$, which is programmed as a variable, but accessed like a dimension.

The word ‘æther’ is an allusion to the luminesiferous æther that was the basis for nineteenth century research in electromagnetism. Reference [6] discusses the æther in detail: suffice it to say that it creates an immersive context in which all calculations ‘bathe’, like the water flowing through biological cells.
The idea is that at time \( t \), \#ÆTHER will simply evaluate the variable ÆTHER with no context information apart from the current value of time. Since, ÆTHER is a variable, it can have subvariables, and these can have subvariables, so that this runtime context can have as much substructure as is necessary.

During instant \( t \), during the evaluation of expressions, some new equations for ÆTHER can be generated. These do not contribute to the bestfitting for ÆTHER at instant \( t \); they only become relevant at instant \( t + 1 \).

Using this approach, the æther can be programmed jointly by all components of a system as well as from the outside. The bestfitting inherent in the way that TransLucid works ensures that if multiple definitions are made, there will be some meaningful outcome, one of the possibilities being `sp<multidef>`.

Note that the timing of the ÆTHER is of the enclosing system not of the internal systems. As a result, the changes to the ÆTHER are slow in comparison to the timings within the internal systems. There is therefore no problem with conceptually centralizing this information, as it can readily be distributed if necessary.

More decentralized sharing of context between component systems requires developing more complex protocols. Nevertheless, this should not be too difficult, since each component acts like a synchronous reactive system.

6 Conclusions

The general aim of the project is to extend the applicability of Cartesian programming to context-aware distributed systems. We discuss below the implications.

Distributed systems can be described in TransLucid by using another physical dimension: node. Using a TransLucid system of equations that varies with time allows both the programming of reactive systems as well as systems whose behavior can be corrected over time. Allowing variation with respect to the node enables the definition of physically distributed systems. These two dimensions have a physical interpretation when this is appropriate, but their interface is identical to that of other dimensions.

The simplest approach would be a centralized and synchronous interpretation for the node dimension; however, this approach would violate the very principle of distribution. Hence we need to develop both a realistic semantics and effective methods of distribution, supposing a dynamic physical architecture.

If we consider a synchronous TransLucid system, at each instant \( t \in \mathbb{N} \), there is a system of equations \( Q_t \) and a demand \((E_t, \kappa_t)\) (possibly several). The evaluation of this demand could take place by using several execution threads that would interact by registering intermediate calculated values in a cache. This cache would therefore include equations of the form \( x \mid \kappa = E \), i.e., their form would be identical to that of an ordinary TransLucid equation; in fact, it is a TransLucid equation. It can thus be said that evaluating a TransLucid program takes place by writing another TransLucid program, equivalent but allowing quicker access to already computed values. Hence, within each instant \( t \in \mathbb{N} \), there exists a sequence of sets of equations \( Q_{t0}, \ldots, Q_{ti}, \ldots \) such that \( Q_{ti} \subseteq Q_{tj} \) for all \( i \leq j \). We therefore use an infinitesimal time describing the infinitely faster (conceptual) evolution within an instant \( t \) with respect to the evolution between instants. Each thread has its own independent infinitesimal clock that is adjusted forward whenever it must communicate with another thread (or with the cache) whose clock is further advanced.

The technique of writing a TransLucid program while evaluating a TransLucid program provides not only the means for implementing the language but also a model for the writing of distributed TransLucid programs. Sending a message in an object-oriented system becomes the sending of a new equation or a new demand to be evaluated by another TransLucid system. By using the same temporal semantics for the evaluation of a TransLucid demand, a purely declarative semantics can be derived. Each shipment by node \( p_1 \), tagged by clock instant \( t_1 \), to node \( p_2 \) is immediately tagged at \( p_2 \) with its current instant \( t_2 \). The exact value of \( t_2 \) depends on the behavior of the network and hence is outside the control of both \( p_1 \) and \( p_2 \); however, once the
choice of $t_2$ is made, the rest of the behavior is deterministic and replicable. Should another demand tagged $(p_1, t_1)$ arrive at $p_2$, then the set of equations available at $(p_2, t_2)$ would be used. It then becomes possible to describe a distributed system whose collective semantics can be derived simply.

References


