The Practical Uses of TransLucid

John Plaice
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW, 2052
Australia
plaice@cse.unsw.edu.au

Blanca Mancilla
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW, 2052
Australia
mancilla@cse.unsw.edu.au

ABSTRACT

TransLucid is a declarative coördination language in which all expressions vary according to an arbitrarily-dimensional context. Using only the concepts of context change and of context query, a wide variety of programming constructs are demonstrated to be easily programmed in TransLucid. The language can therefore be used in its own right or as a target language for different paradigms.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—
Language Constructs and Features

General Terms

Languages

Keywords

Cartesian Programming, Lucid language, declarative programming, multidimensional programming, context-aware programming.

1. INTRODUCTION

In this article we show how the TransLucid multidimensional programming language [5] can be used to solve a wide variety of problems in a declarative manner. In TransLucid, expressions vary in a multidimensional context, where any ground value may play the rôle of a 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.

We call the key structure in TransLucid a *hyperdaton*, an arbitrary-dimensional infinite array, indexed by a multidimensional context in the form of a dynamically generated lazy tuples, i.e., sets of (*dimension*, *value*) pairs.

The denotational and operational semantics of TransLucid have been presented in [5]. Extensions of the language to support reactive, parallel and distributed programming are presented in [4]. An approach to implementating TransLucid on a multithreaded processor is given in [7]. The history leading to the development of TransLucid, ultimately going back to Wadge and Ashcroft's Lucid language [2], can be found in [6].

In this paper, we use a number of examples to demonstrate how TransLucid can be used as a programming language in its own right, as a coördination language, and as a target language for a variety of paradigms, potentially adding context-awareness to these for free. The examples given are simple recursive functions, context-aware constants and operations, unlimited register machines, functional programs and context-aware programs.

CASTA'09, August 24, 2009, Amsterdam, The Netherlands.

2. RECURSIVE FUNCTIONS

We begin the presentation of TransLucid using a simple, commonly understood example: the recursive function. The following program defines the Fibonacci, factorial and Ackermann functions, and then makes three demands.

```
01  infixn "==" "operator==" 1;;
02  infixl "+" "operator+" 2;;
03  infixl "-" "operator-" 2;;
04  infixl "*" "operator*" 3;;
05  %%
06  fib  = if #0 _==_ 0 then 1
07         elsif #0 _==_ 1 then 1
08         else fib@[0:#0-2] + fib@[0:#0-1]
09         fi;;
10  fact = if #0 _==_ 0 then 1
11         else #0 * fact@[0:#0-1]
12         fi;;
13  ack  = if #0 _==_ 0 then #1+1
14         elsif #1 _==_ 0
15         then ack@[0:#0-1,1:1]
16         else ack@[0:#0-1,1:ack@[1:#1-1]]
17         fi;;
18  %%
19  fib@[0:4];;
20  fact@[0:4];;
21  ack@[0:3,1:4];;
```

The example consists of three parts, the header (01–04), the equations (06–17) and the demands (19–21).

The header provides sufficient information so that the equations and the demands can be properly parsed. The header defines four operators, `==`, `+`, `-` and `*`. Should one of these operators be recognised by the parser, the latter

maps the operator name to a function name. The function name is then used to find a function defined in the underlying host language (currently C++). Each of the header lines also designates the associativity and precedence of the operators.

TransLucid has four ordinary builtin types: Boolean, integer of arbitrary precision, Unicode character and Unicode string. These can be parsed and printed without taking the context into account. As a result, almost any Unicode character can be used to create operators. Single-character operators can be used directly in expressions; multiple-character operators must be placed inside pairs of underscores, as in `_==_`.

In the expressions, the `#` operator is used to query the current context; for example, on line `06`, subexpression `#0` means the current value of dimension `0`. As for the `@` operator, it is used to change the current context; for example, on line `08`, subexpression `fib@[0:#0-2]` means to recurse and reëvaluate `fib` having changed the context by replacing the current value for dimension `0` by decrementing it by 2.

Context changes are *relative*, meaning they only change the values of dimensions that are explicitly mentioned. For example, in line `16`, subexpression `ack@[1:#1-1]` changes the setting for dimension `1`, but does not change the setting for dimension `0`, or any other should there be one.

Runnning the above program yields the following results.

```
5 ;;
24 ;;
125 ;;
```

3. CONTEXT-DEPENDENT CONSTANTS

TransLucid has been designed so that externally defined types can be used as atomic types in TransLucid. For each type, there is a parse function, which transforms a string into an object of that type, and a print function, which generates a string from an object of that type. Of course, because TransLucid is a context-aware language, at *every* level, even the parsing and printing of constants is context-dependent. Therefore, for each data type, there may be predefined dimensions of variance.

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.

```
01  library "si";;
02  library "verb";;
03  %% %%
04  1000000000000000000000000000000000 @
05    [outintmp:true, outtext:true,
06     outeunames:true] ;;
07  si<Hz> @ [instyle:"symbol", outstyle:"name"];;
08  si<newton> @
09    [instyle:"name", outstyle:"basicunits"];;
10  si<electric capacitance> @
11    [instyle:"quantity", outstyle:"name"];;
12  verb<envoyer> @
13    [outmode:"indicative", outtense:"future",
14     outperson:"1s"];;
15  verb<assiègeai> @
16    [inmode:"indicative", intense:"past",
17     inperson:"1s"];;
```

Lines `04`–`06` use three predefined dimensions relevant for printing integers: `outintmp` means to use the sophisticated printer, `outtext` means to print the number textually, and `outeunames` means to use the long scale for printing numbers. The result for that demand is:

```
intmp<one trilliard> ;;
```

Lines `07`–`11` involve the type `si`, imported using line `01`, used for manipulating SI units. That library also defines dimensions `instyle` and `outstyle`. The results of the demands are:

```
si<hertz> ;;
si<m.kg.s-2> ;;
si<farad> ;;
```

Lines `12`–`17` involve the type `verb`, imported using line `02`, used for conjugating French verbs. That library defines dimensions `inmode`, `intense`, `inperson`, `outmode`, `outtense` and `outperson`. The first demand conjugates an infinitive and the second demand computes the infinitive. Here are the results:

```
verb<enverrai> ;;
verb<assiéger> ;;
```

In most situations, the internal representation will have a canonical form. This is certainly the case for the above example. The integers are held as GNU MP integers, the SI units are held as 7-tuples of integers (m, kg, s, A, K, mol, cd) and the verbs are held in their infinitive form. However, having a canonical representation is by no means necessary, and may even be impossible.

4. TURING COMPLETENESS

Given that TransLucid can be used to write recursive programs, it is clear that it is Turing-complete. It turns out that TransLucid can be viewed as a declarative form of the well-known model of computation of *unlimited register machines* (URM) [3], also known as *random access machines* (RAM).

An unlimited register machine contains an infinite set of registers, called R_1, R_2, ..., R_n, etc. A program running on this machine is a finite sequence of instructions I_1, I_2, ..., I_m, where the instructions may be of the form:

- `Z`(i): Set register R_i to 0, next instruction.
- `S`(i): Add 1 to register R_i, next instruction.
- `T`(i, j): Copy register j into register i, next instruction.
- `J`(i, j, k): If registers i and j are equal, proceed to instruction k, otherwise proceed to the next instruction.

The way a URM program works is that if there are n input values, they are placed in registers R_1 through R_n, the program is then started with instruction 1, and the program will stop the instant that the next instruction number is greater than m. The answer is left in register R_1.

Given a URM program $P = (I_1, \ldots, I_m)$, it can be easily translated into TransLucid, using a single variable, say X, defined recursively. The idea is that dimension 0 will be used to keep track of the URM instruction counter, while dimension j, $j \geq 1$, is used to keep track of register R_j.

$$\begin{aligned} X \mathbin{@} [0 : i] &= \mathcal{T}(I_p), \quad p = 1..m \\ X &= \#1 \end{aligned}$$

where each $\mathcal{T}(I_p)$ means the appropriate translation of instruction I_p:

$$\begin{aligned}
\mathcal{T}(\mathtt{Z}(i)) &= X \mathbin{@} [0 : \#0 + 1,\ i : 0] \\
\mathcal{T}(\mathtt{S}(i)) &= X \mathbin{@} [0 : \#0 + 1,\ i : \#i + 1] \\
\mathcal{T}(\mathtt{T}(i,j)) &= X \mathbin{@} [0 : \#0 + 1,\ i : \#j] \\
\mathcal{T}(\mathtt{J}(i,j,k)) &= \mathtt{if}\ \#i = \#j \\
&\quad \mathtt{then}\ X \mathbin{@} [0 : k] \\
&\quad \mathtt{else}\ X \mathbin{@} [0 : \#0 + 1]\ \mathtt{fi}
\end{aligned}$$

If the input consists of values $(i_1, \ldots, i_n)$, then the initial demand is:

$$\mathtt{X} \mathbin{@} [0 : 1,\ 1 : i_1,\ \ldots,\ n : i_n]$$

For example, the following program will, assuming $a \geq b$, $R_1 = a$ and $R_2 = b$, calculate $a - b$:

```
1: J(1,2,5)
2: S(2)
3: S(3)
4: J(1,1,1)
5: T(3,1)
```

Register R_2 is incremented up to the value held in register R_1, namely a. In parallel, R_3 is incremented up to $a - b$. Once registers R_1 and R_2 are equal, the value in register R_3 is placed in register R_1 and the program halts. Note that instruction 4, `J(1,1,1)`, is an unconditional branch to instruction 1, since register R_1 is always equal to itself.

The TransLucid solution uses context-dependent definitions:

```
01  infixn "==" "operator==" 1;;
02  infixl "+" "operator+" 2;;
03  %%
04  sub @ [0:1] = if #1 _==_ #2 then sub @ [0:5]
05                else sub @ [0:#0+1] fi;;
06  sub @ [0:2] = sub @ [2:#2+1, 0:#0+1];;
07  sub @ [0:3] = sub @ [3:#3+1, 0:#0+1];;
08  sub @ [0:4] = if #1 _==_ #1 then sub @ [0:1]
09                else sub @ [0:#0+1] fi;;
10  sub @ [0:5] = sub @ [1:#3, 0:#0+1];;
11  sub         = #1
12  %%
13  sub @ [0:1, 1:4, 2:3] ;;
```

The result is below:

```
1 ;;
```

5. FUNCTIONAL ABSTRACTION AND WHERE CLAUSES

It is common practice in all programming languages to have structuring. To enhance structuring in TransLucid, non-recursive functions can be transparently added since one only needs variable substitution; recursive functions are handled in the next section. In TransLucid, we implement the *indexed where clause*, first used in Indexical Lucid [1]:

```
E
where
  index d1, ..., dn;
  x1 = E1;
  ...
  xn = En;
end where;
```

The TransLucid indexed where clause is not primitive. Rather, it is implemented by translating it into a series of context-dependent definitions that suppose an indexing of the different where clauses. Suppose that clause w appears at nesting level m, i.e., there are $m - 1$ surrounding where clauses. Then we will need m dimensions, $\mathtt{w}_1$ to $\mathtt{w}_m$, to keep track of all of the levels of definitions. Then the contents of the above clause can be rewritten as:

$$\begin{aligned}
x_1 \mathbin{@} [\mathtt{w}_i = k_i]_{i=1..m} &= \mathcal{T}(E_1, 1 : \ell) \\
&\ldots \\
x_1 \mathbin{@} [\mathtt{w}_i = k_i]_{i=1..m} &= \mathcal{T}(E_n, n : \ell)
\end{aligned}$$

where $\mathcal{T}(E, \ell)$ does the appropriate translation. As for the demand, if $\ell = \langle k_1, \ldots, k_m \rangle$, it is translated to:

$$\mathcal{T}(E, \ell) \mathbin{@} [\mathtt{w}_m : k_m]$$

The internal dimensions are not visible outside the parentheses, nor are the definitions of `x1` through `xn`. The definitions in all of the enclosing where clauses are visible, but with lower: priority than the current ones.

6. FUNCTIONAL PROGRAMMING

In the final section of *Lucid, the Dataflow Programming Language* [9], Wadge and Ashcroft explained that the addition of functions to Lucid was problematic, because Lucid objects are infinite. One would like to apply higher-order functions to infinite objects, to build infinite objects containing higher-order functions, and even to build infinite objects containing higher-order functions that can be applied to infinite objects.

With TransLucid, it is possible to provide a general solution to this problem. It is based on the idea that an n-argument function can be understood as an (at least) n-dimensional hyperdaton, and that a function call corresponds to looking up the value of that hyperdaton in the appropriate context. This idea clearly works simply for first-order functions, but how can it be extended to deal with higher-order functions and partially applied functions?

Higher-order functions must be able to manipulate other functions as if they are single values. To do this in TransLucid, there is a single hyperdaton called `F`, which varies in the `fun` dimension. By changing the value of the `fun` dimension, we change the function.

For partially applied functions, we need to explicitly pass around tuples encapsulating the partially applied functions. To do this, we need to introduce two new dimensions: `args` is the number of arguments in the function, while `count` is the number of accumulated arguments.

For example, in

```
add 3 4
where
  add x y = x+y ;;
end where;;
```

we build successively:

```
[fun:"add", args:2, count:0]
[fun:"add", args:2, count:1, 1:3]
[fun:"add", args:2, count:2, 1:3, 2:4]
```

and then a lookup into `F` takes place.

The general solution is to replace every function call of the form `E1 E2` with `Ap(E1,E2)`, defined below:

```
Ap(E1,E2) =
  if a+1 _==_ c then F @ ctxt else ctxt fi
  where
    a = #args @ E1;;
    b = a + 1;;
    c = #count @ E1;;
    ctxt = E1 . [args:b, b:E2];;
  end where;;
```

We give an example from Panagiotis Rondogiannis [8]:

```
f 8
where
  f x       = twice (add x) x
  twice g y = g (g y)
  add a b   = a + b
end where;
```

here translated into TransLucid:

```
Ap([fun:"f", args:1, count:0], 8)
where
  F @ [fun:"f"]     =
    Ap(Ap([fun:"twice", args:2, count:0],
          Ap([fun:"add", args:2, count:0],
             #1)),
       #1);
  F @ [fun:"twice"] = Ap(#2, Ap(g, #1));
  F @ [fun:"add"]   = #2 + #1
end where;
```

We can see from this process that identifiers themselves are not truly necessary. We could have a single identifier, say `H`, and then all others would simply be coded as variance of `H` in the `id` dimension.

7. HYPERDATONS OF FUNCTIONS

The very last example of the Lucid book posited that one could build streams of functions, using lambda expressions. Up to now, no language had been able to do this until TransLucid. We even go one level further, by allowing entire *hyperdatons of functions*, i.e., infinite multidimensional families of functions. We simply index all of the occurrences of λ. For example,

```
01  dimension "d";;
02  %%
03  pow = if #d _==_ 0 then \x : x
04        else \x : x * (pow@[d:#d-1])(x) fi;
05  odd = if #1 _==_ 0 then 1
05        else odd@[d:#d-1] + 2 fi;
06  %%
07  (pow@[d:4])(odd@[d:3]);;
```

would yield the result $7^5 = 16807$.

8. CONTEXT-AWARE PROGRAMMING

What is commonly called "context-aware programming" corresponds to the use of an "external context" to which a program must adapt as the former changes. In TransLucid, the "external context" is simply a TransLucid context, and the "external context changes" means that the corresponding TransLucid context varies according to a dimension with *physical* interpretation, say `time`. Using the synchronous hypothesis, all variables vary according to that dimension.

Depending on the deployment environment of a TransLucid system, the exact mechanism for reading the "external context" will vary. However, whatever the mechanism, the semantics does not change.

In the following example, the system of equations defines a new external context, called `contextout`, based on an external context called `contextin`. The variable `contextout` is the same as `contextin`, except for dimension `x`, which doubles `x` from the previous instant, except for the first.

```
01  %%
02  contextin = external ;;
03  contextout = contextin @
04    if #time == 0 then []
05    else [x : (#x * 2) @ [time : #time-1]] fi;;
06  %%
07  contextout ;;
```

The output `contextout` is a `time`-varying entity, so the above program defines a reactive system.

Since TransLucid is, from the core, a language for manipulating context, adapting TransLucid for "context-aware programming" is a simple issue.

9. CONCLUSION

The examples of TransLucid given in this article amply demonstrate the versatility of the language, both as host language and as target language for other paradigms. The concept of arbitrarily-dimensional context can be used to encode many different forms of computing. Current research is focused on the structuring of data, on developing support for reactive and distributed programing, and for the declarative handling of side-effects.

10. REFERENCES

[1] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional Programming*. Oxford University Press, New York, 1995.

[2] E. A. Ashcroft and W. W. Wadge. Lucid, A Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[3] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[4] B. Mancilla and J. Plaice. Declarative multithreaded programming. In *33rd IEEE International Computer Software and Applications Conference*, 2009. In press.

[5] J. Plaice and B. Mancilla. Cartesian programming: The TransLucid programming language. In *33rd IEEE International Computer Software and Applications Conference*, 2009. In press.

[6] J. Plaice, B. Mancilla, and G. Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Mathematics in Computer Science*, 2(1):63–84, 2008.

[7] T. Rahilly and J. Plaice. A multithreaded implementation for TransLucid. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 1272–1277, 2008.

[8] P. Rondogiannis. Personal communication, 2005.

[9] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.