SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:609–630 (DOI: 10.1002/spe.582)

GLU♮ embedded in C++: a marriage between multidimensional and object-oriented programming

Nikolaos S. Papaspyrou[1,*,†] and Ioannis T. Kassios[2]

[1]*School of Electrical and Computer Engineering, Software Engineering Laboratory, National Technical University of Athens, Polytechnioupoli, 15780 Zografou, Athens, Greece*
[2]*Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, M5S 3G4 Ontario, Canada*

SUMMARY

The embedding of a small but expressive language of multidimensional functional programming in a well known and widely used language of object-oriented programming leads to the combination of two radically different programming models. In this paper, we formally define the syntax and semantics of GLU♮, which can be thought of as the multidimensional core of Lucid and GLU, and we describe its implementation as a language embedded in C++. With the aid of a few examples, we argue that the marriage of the two programming models is not only compatible and natural, but also that it produces a new and interesting hybrid language. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: multidimensional programming; object-oriented programming; GLU; C++; lazy arrays; lazy functions

1. INTRODUCTION

Multidimensional languages are particularly suited for programming dynamic systems whose state varies in one or more dimensions. Temporal and dataflow languages are particularly useful and well studied members of this family. Multidimensional languages are based on intensional logic [1] and typically adopt characteristics of other functional or logic programming languages. *Lucid*, designed by Ashcroft and Wadge [2,3], started as a simple, non-imperative temporal programming language but

*Correspondence to: Nikolaos S. Papaspyrou, School of Electrical and Computer Engineering, Software Engineering Laboratory, National Technical University of Athens, Polytechnioupoli, 15780 Zografou, Athens, Greece.
†E-mail: nickie@softlab.ntua.gr

Contract/grant sponsor: Greek Secretariat for Research and Technology, under the programme ΠΕΝΕΔ'99. Project code 99EΔ265

Received 16 July 2001
Revised 12 September 2003
Accepted 12 September 2003

Published online 24 February 2004

soon developed into an inherently parallel multidimensional programming language [4]. Its descendant *GLU* (Granular Lucid) emphasizes the parallel programming aspects of multidimensionality [5,6]. A large number of publications in the last two decades describe the two languages and present a wide variety of applications in areas such as real-time applications [7,8], scientific programming [9,10], temporal databases [11], multidimensional spreadsheets [12], digital signal processing [13], attribute grammars [14], version control [15] and Internet publishing [16].

Although a promising approach, multidimensional programming has not appealed to the community of imperative programming. The two paradigms interface very poorly: on the one hand conventional imperative programming languages have no room for multidimensionality; on the other, existing multidimensional languages cannot take advantage of imperative features and techniques, developed over years of research, that typically result in much better performance. A convergence of the two paradigms would be beneficial for software developers, provided that the virtues of the two were to be respected.

Taking into account the popularity and success of imperative programming languages, which indisputably have the biggest part of the pie in software development, embedding multidimensional features in them seems to be a particularly suitable way to attempt this convergence. In this way, it is possible to reuse the syntax, semantics, implementations, libraries, tools and in general the whole infrastructure of an existing, widely spread programming language and in addition to use the embedded multidimensional features to facilitate the programming of specialized applications. The same approach has been proposed for the implementation of *domain specific languages* (DSLs), which have recently attracted the interest of many researchers from many different scientific areas [17]. Although multidimensional languages have a much wider domain of applications than a typical DSL, the two cases present many similarities.

Following this line of thought, in this paper we propose the embedding of a concise multidimensional language, which we call GLU$^\natural$, in C++. The language GLU$^\natural$ consists of the core of multidimensional features that are present in Lucid and GLU. It can be considered as a language orthogonal to C++ and is implemented as a collection of C++ classes and class templates. The syntax and semantics of C++ remain unchanged. Programmers, however, are able to use multidimensional objects which can take the form of lazy arrays and lazy functions.

The choice of C++ as the host language is a natural one. It is a very popular object-oriented programming language with numerous applications and a large amount of existing libraries and tools. Furthermore, C++ is especially suited for embedding DSLs. This is partly due to the expressiveness of the object-oriented programming model and partly to some particular characteristics of C++, such as parametric polymorphism with the use of templates, exception handling, operator overloading and the presence of a powerful preprocessor.

A similar embedding of multidimensional characteristics in a conventional programming language has been proposed by Panos Rondogiannis [18]. In his approach, Java is used as the host language; however its syntax and semantics are slightly changed and a custom preprocessor is needed to translate the embedded programs into pure Java code. Furthermore, the embedded language supports only a subset of the dimensional operators of GLU$^\natural$, multidimensionality is limited to zero-order objects (i.e. streams or lazy arrays) and some additional restrictions are imposed, due to the details of his implementation. Our approach is clearly in the same direction. It contributes by removing the aforementioned shortcomings, resulting in a deeper embedding of multidimensionality in an object-oriented programming language. Furthermore, our approach uses dimensionality analysis to

calculate the maximal set of dimensions in which the value of an expression varies, leading to a much more efficient implementation.

The rest of the paper is structured as follows. Section 2 contains a brief introduction to multidimensional programming and GLU, as a provision for the unaware reader. In Section 3 we formally define GLU$^{\natural}$. In Section 4 we describe the embedding of GLU$^{\natural}$ in C++ from the programmer's point of view. Implementation details of this embedding are given in Section 5. Section 6 illustrates the convergence of the two paradigms with some characteristic examples of GLU$^{\natural}$/C++ programs. The paper ends with a premature evaluation of our approach in Section 7 and a few concluding remarks.

2. MULTIDIMENSIONAL PROGRAMMING

In multidimensional languages like GLU, the values of expressions depend on an implicit context that is captured by a set of *dimensions*. Consider an expression which is meant to provide the temperature in degrees Celcius. In a typical imperative program, this expression could have the numerical value 25. However, in the real world it is the case that temperature varies in at least two ways: (i) between different locations in space, and (ii) between different moments in time. The numerical value 25 only captures the temperature at a specific place and time. The specific value of an expression in a given context is called the expression's *extension*. However, in a GLU program the same expression could be viewed as a multidimensional entity, varying in two dimensions: space and time. In this way, its value in some context (e.g. in Athens in the month of May) could be 25, but in another context (e.g. in midwinter Siberia) it could be -40. The collection of all possible extensions of an expression in all different contexts is called the expression's *intension*. Intensions are commonly the principal type of data values in multidimensional programming languages.

There can be as many dimensions as a program requires. The value of a given expression may vary in some of these dimensions, while remaining constant in others. In Lucid and GLU, dimensions are discrete and linear: points along a dimension are indexed by the natural numbers 0, 1, 2, etc[‡]. A set of *dimensional operators*, like `first`, `next` and `fby`, can be used to switch between different points. In GLU, dimensional operators are tagged with the dimension upon which they act. To understand all this, let us see a few examples. The following line defines the infinite stream of natural numbers along dimension `t`. (Parentheses are not required, but they are used here and elsewhere in this paper for reasons of clarity.)

```
nat = 0 fby.t (nat + 1) ;
```

Translated in words, this equation says that the first value of `nat` along dimension `t` is 0 and that each next value in the sequence can be computed by increasing the previous value by 1. Using an explicit subscript for the context in which `nat` is evaluated, this means that

$$\mathtt{nat}_{\mathtt{t}=0} = 0$$
$$\mathtt{nat}_{\mathtt{t}=i+1} = \mathtt{nat}_{\mathtt{t}=i} + 1 \quad \text{for all } i \geq 0$$

[‡]Another possibility (not discussed in this paper) is to have *branching dimensions*, which are tree-like in nature. An example of a temporal programming language based on branching-time logic is *Cactus* [19].

The dimensional operator `fby.t`, read 'followed by, in dimension `t`', allows us to suppress the explicit dimension indices.

As a second example, consider the following program which defines the infinite stream `fib` of Fibonacci numbers along dimension `t`, using an auxiliary stream `g`.

```
fib = 0 fby.t g ;
g   = 1 fby.t (g + fib) ;
```

When `t` is 0 we have `fib` = 0 and `g` = 1. When `t` is 1, the value of `fib` is equal to the value of `g` when `t` is 0, i.e. 1. The value of `fib` when `t` is 2 is equal to the value of `g` when `t` is 1, which is in turn equal to the sum of the values of `g` and `fib` when `t` is 0, i.e. 0 + 1 = 1.

GLU supports a rich set of dimensional operators. Operators `first.t` and `next.t` are in a sense the inverses of `fby.t`: the former switches to point 0 in the given dimension, while the latter switches to the next point relative to the current one. For example, assume that in the current context `t` is 41. Then in the current context, `nat` = 41, while `first.t nat` = 0 and `next.t nat` = 42. Operator `@.t`, read as 'at, in dimension `t`' switches to the point in dimension `t` that is given by its right operand. Thus, in any context we have `nat @.t 42` = 42 and `fib @.t 6` = 8. By contrast, the operator `#.t`, read as 'index of dimension `t`' does not require any operands: it simply returns the current point in dimension `t`.

Operators `asa.t` and `wvr.t` are the most complex dimensional operators of GLU. The former, read 'as soon as, in dimension `t`', returns the value of its left operand at the first point in dimension `t` where the right operand evaluates to true. For example, `fib asa.t (fib > 25)` = 34, i.e. the first Fibonacci number that exceeds 25. Operator `wvr.t`, read 'whenever, in dimension `t`', selects from its left operand only the values at those points in dimension `t` where the right operand evaluates to `true`. For example, `fib wvr.t (fib % 2 == 0)` produces the infinite stream of even Fibonacci numbers: 0, 2, 8, 34, 144, ...

GLU also supports higher-order functions, with the restriction that their results cannot be of function type. In general, functions in GLU accept a number of data parameters and a number of dimension parameters. The former resemble parameters in a functional language with lazy evaluation, except for their multidimensional nature. The latter are used to abstract over dimensions. Let us clarify this with an example of a function that rotates the `d1` dimension of a stream `x` into its `d2` dimension:

```
realign.d1,d2 (x) = x @.d1 #.d2 ;
```

In this definition, `d1` and `d2` are dimension parameters and `x` is a data parameter. The result of `realign.t,z(fib)` is the infinite stream of Fibonacci numbers ordered along dimension `z` instead of dimension `t`.

Expressions may also contain local definitions with the use of `where` clauses. As a last example, let us see a function which computes the running average of a given stream `x` in a dimension `d`. This definition uses three auxiliary local streams: `mean`, `diff` and `count`.

```
runningAverage.d (x) = mean
   where mean  = x fby.d (mean + diff) ;
         diff  = (next.d x - mean) / (next.d count) ;
         count = 1 fby.d (count + 1) ;
   end;
```


$$
\begin{array}{lll}
P & ::= & D^{*}\ E \\
D & ::= & \tau\ v\ \texttt{=}\ E\ \texttt{;} \\
\tau & ::= & \texttt{real} \mid \texttt{bool} \\
E & ::= & v \mid n \mid \texttt{true} \mid \texttt{false} \mid \mathit{unop}\ E \mid E\ \mathit{binop}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \mid \texttt{\#.}d \\
 & \mid & \texttt{first.}d\ E \mid \texttt{next.}d\ E \mid E\ \texttt{fby.}d\ E \mid E\ \texttt{@.}d\ E \mid E\ \texttt{asa.}d\ E \mid E\ \texttt{wvr.}d\ E \\
\mathit{unop} & ::= & \texttt{-} \mid \texttt{!} \\
\mathit{binop} & ::= & \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{+} \mid \texttt{-} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

Figure 1. Abstract syntax of GLU♮.

3. DEFINITION OF GLU♮

GLU♮ is a small subset of GLU supporting most of its dimensional operators. In addition, it encompasses a lazy expression language with two basic data types (real and Boolean) and a primitive language of recursive definitions. The abstract syntax of the language is given in Figure 1. A program (P) in GLU♮ is a sequence of definitions (D) followed by an expression (E) that must be evaluated. The nonterminals τ, *unop* and *binop* stand for types, unary and binary operators, respectively. Variable identifiers are denoted by v, dimension identifiers by d and numeric constants by n.

Although GLU♮ is of functional origin, we should emphasize that it does not support functions directly. The introduction of functions (even of higher order) would not complicate the syntax and semantics of the language significantly. However, their presence is not necessary since GLU♮ is intended as a language embedded in C++, which provides a mechanism for defining functions. Supporting lazy first-class higher-order functions directly in GLU♮ would be hard to implement efficiently in C++. Moreover, previous results suggest that the omission of functions from a multidimensional language does not harm it much, in terms of expressiveness [20,21]. Notice that C++'s functions are at least as general as those of GLU, which does not support unnamed functions (lambda abstractions) and currying. Furthermore, C++'s templates can be used to create polymorphic functions, which is not possible in GLU.

For the formal definition of the semantics of GLU♮ we use the denotational approach [22,23]. We distinguish between static, typing and dynamic semantics.

3.1. Static semantics

The basic domains **Var** and **Dim** contain variable and dimension identifiers. Two additional domains, **Type** and **Ent** represent data types and type environments, respectively. As an abuse of notation, we do not distinguish between elements of the syntactic classes v, d and τ and the elements of the corresponding semantic domains **Var**, **Dim** and **Type**.

$$\tau : \mathbf{Type} = \{\text{real}, \text{bool}\}$$
$$\Gamma : \mathbf{Ent} = \mathcal{P}(\mathbf{Var} \times \mathbf{Type})$$

$\mathcal{P}(A)$ is the powerdomain of A. Intuitively a type environment is a set of pairs, each defining the type of a program variable. By Γ_0 we represent the empty type environment:

$$\Gamma_0 = \emptyset$$

The purpose of static semantics is to define the type environment that corresponds to the whole program. Thus, the static meanings of definitions and sequences of definitions, denoted as $\{\!|\ D\ |\!\}$ and $\{\!|\ D^*\ |\!\}$, respectively, are type environments. An empty sequence of definitions produces the empty type environment. The powerdomain union operator is used to concatenate two type environments in adjacent definitions.

$$\{\!|\ \epsilon\ |\!\} = \Gamma_0$$
$$\{\!|\ DD^*\ |\!\} = \{\!|\ D\ |\!\} \cup \{\!|D^*|\!\}$$

The type environment corresponding to a single definition contains a single pair, which is produced by using the powerdomain singleton operator.

$$\{\!|\tau\, v{=}E;\ |\!\} = \{\langle v, \tau\rangle\}$$

3.2. Typing semantics

The typing semantics of GLU$^\sharp$ associates types with syntactic elements of GLU$^\sharp$ and detects typing errors in programs. It is defined using typing judgements of the form $\Gamma \vdash p : \theta$, where Γ is a type environment, p is a program phrase (i.e. a part of the program according to abstract syntax) and θ is a phrase type. In particular, θ can be one of the following: prog$[\tau]$, defn, or exp$[\tau]$. The first and third alternatives represent programs or expressions that compute results of type τ. The second represents definitions or sequences of definitions. The complete typing semantics for GLU$^\sharp$ is presented in Figure 2 using a set of inference rules.

3.3. Dynamic semantics

The dynamic semantics of GLU$^\sharp$ is given in Figure 3. The domains **T**, **N** and **R** denote respectively the flat domains of truth values, natural and real numbers. The elements of domain **V** are simple values of the GLU$^\sharp$ data types. In the standard way of defining the semantics of multidimensional languages, the elements of **W** represent *possible worlds*, mapping each dimension to a natural number giving the dimension's present index. Domain **D** contains the denotations of expressions, which are functions from possible worlds to simple values. In this way, an expression's value is multidimensional. Also, the elements of domain **Env** are environments mapping the defined variables to their denotations. The distinguished elements w_0 : **W** and ρ_0 : **Env**, used as the initial world and environment, are constant functions with a value of $\perp$. Bottom elements are used to model both runtime errors (e.g. division by zero) and nontermination.

In the semantic functions defining the dynamic semantics of GLU$^\sharp$ in Figure 3, the least fixed point operator is required in the denotation of programs to allow recursive definitions. If f is a function, we use the notation $f\{a \mapsto b\}$ to denote a function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \neq a$. The denotations of unary and binary operators are straightforward and have been omitted.

Programs and definitions

$$\frac{\Gamma = \{\!|D^*|\!\} \quad \Gamma \vdash D^* : \text{defn} \quad \Gamma \vdash E : \exp[\tau]}{\vdash D^*\ E : \text{prog}[\tau]}$$

$$\frac{}{\Gamma \vdash \epsilon : \text{defn}} \qquad \frac{\Gamma \vdash D : \text{defn} \quad \Gamma \vdash D^* : \text{defn}}{\Gamma \vdash D\ D^* : \text{defn}} \qquad \frac{\Gamma \vdash E : \exp[\tau]}{\Gamma \vdash \tau\ v\ \texttt{=}\ E\ \texttt{;} : \text{defn}}$$

Simple expressions

$$\frac{\langle v, \tau\rangle \in \Gamma}{\Gamma \vdash v : \exp[\tau]} \qquad \frac{}{\Gamma \vdash n : \exp[\text{real}]} \qquad \frac{}{\Gamma \vdash \texttt{true} : \exp[\text{bool}]}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \exp[\text{bool}]} \qquad \frac{\Gamma \vdash E : \exp[\text{real}]}{\Gamma \vdash \texttt{-}\ E : \exp[\text{real}]} \qquad \frac{\Gamma \vdash E : \exp[\text{bool}]}{\Gamma \vdash \texttt{!}\ E : \exp[\text{bool}]}$$

$$\frac{\Gamma \vdash E_1 : \exp[\text{real}] \quad \Gamma \vdash E_2 : \exp[\text{real}] \quad \mathit{binop} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash E_1\ \mathit{binop}\ E_2 : \exp[\text{real}]}$$

$$\frac{\Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\tau] \quad \mathit{binop} \in \{\texttt{==}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\}}{\Gamma \vdash E_1\ \mathit{binop}\ E_2 : \exp[\text{bool}]}$$

$$\frac{\Gamma \vdash E_1 : \exp[\text{bool}] \quad \Gamma \vdash E_2 : \exp[\text{bool}] \quad \mathit{binop} \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash E_1\ \mathit{binop}\ E_2 : \exp[\text{bool}]}$$

$$\frac{\Gamma \vdash E : \exp[\text{bool}] \quad \Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\tau]}{\Gamma \vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2 : \exp[\tau]}$$

Expressions with dimensional operators

$$\frac{\Gamma \vdash E : \exp[\tau]}{\Gamma \vdash \texttt{first.}d\ E : \exp[\tau]} \qquad \frac{\Gamma \vdash E : \exp[\tau]}{\Gamma \vdash \texttt{next.}d\ E : \exp[\tau]} \qquad \frac{}{\Gamma \vdash \texttt{\#.}d : \exp[\text{real}]}$$

$$\frac{\Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\tau]}{\Gamma \vdash E_1\ \texttt{fby.}d\ E_2 : \exp[\tau]} \qquad \frac{\Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\text{real}]}{\Gamma \vdash E_1\ \texttt{@.}d\ E_2 : \exp[\tau]}$$

$$\frac{\Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\text{bool}]}{\Gamma \vdash E_1\ \texttt{asa.}d\ E_2 : \exp[\tau]} \qquad \frac{\Gamma \vdash E_1 : \exp[\tau] \quad \Gamma \vdash E_2 : \exp[\text{bool}]}{\Gamma \vdash E_1\ \texttt{wvr.}d\ E_2 : \exp[\tau]}$$

Figure 2. Typing semantics of GLU$^\natural$.

The semantics of simple expressions, not involving dimensional operators, are also standard. Non-dimensional operators have a pointwise semantics: the current possible world is propagated to all their operands.

In the denotation of expressions, the most interesting cases are those of the dimensional operators; these are the only equations in which the current possible world w is used. The first five equations are easy; they manipulate w in a simple way, accessing the index of a given dimension and possibly switching to a different point. The equations for the dimensional operators `asa` and `wvr` are the

Domains

$$\begin{aligned} \mathbf{V} &= \mathbf{R} \oplus \mathbf{T} \\ w : \mathbf{W} &= \mathbf{Dim} \to \mathbf{N} \\ \mathbf{D} &= \mathbf{W} \to \mathbf{V} \\ \rho : \mathbf{Env} &= \mathbf{Var} \to \mathbf{D} \end{aligned}$$

Semantics of programs

$$\begin{aligned} &[\![P]\!] : \mathbf{V} \\ &[\![D^* \; E]\!] = \\ &\quad \textbf{let } \rho = \text{fix } (\lambda\rho : \mathbf{Env}.\ [\![D^*]\!]\ \rho\ \rho_0) \\ &\quad \textbf{in } [\![E]\!]\ \rho\ w_0 \end{aligned}$$

Semantics of definitions

$$\begin{aligned} &[\![D^*]\!] : \mathbf{Env} \to \mathbf{Env} \to \mathbf{Env} \\ &[\![\epsilon]\!]\ \rho\ \rho' = \rho' \\ &[\![D\ D^*]\!]\ \rho\ \rho' = [\![D^*]\!]\ \rho\ ([\![D]\!]\ \rho\ \rho') \end{aligned}$$

$$\begin{aligned} &[\![D]\!] : \mathbf{Env} \to \mathbf{Env} \to \mathbf{Env} \\ &[\![\tau\ v = E\ ;]\!]\ \rho\ \rho' = \rho'\{v \mapsto [\![E]\!]\ \rho\} \end{aligned}$$

Semantics of simple expressions

$$\begin{aligned} &[\![E]\!] : \mathbf{Env} \to \mathbf{D} \\ &[\![v]\!]\ \rho\ w = \rho\ v\ w \\ &[\![n]\!]\ \rho\ w = n \\ &[\![\texttt{true}]\!]\ \rho\ w = \text{true} \\ &[\![\texttt{false}]\!]\ \rho\ w = \text{false} \\ &[\![unop\ E]\!]\ \rho\ w = [\![unop]\!]\ ([\![E]\!]\ \rho\ w) \\ &[\![E_1\ binop\ E_2]\!]\ \rho\ w = \\ &\quad [\![binop]\!]\ \langle [\![E_1]\!]\ \rho\ w, [\![E_2]\!]\ \rho\ w \rangle \\ &[\![\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2]\!]\ \rho\ w = \\ &\quad \textbf{if } [\![E]\!]\ \rho\ w \textbf{ then } [\![E_1]\!]\ \rho\ w \textbf{ else } [\![E_2]\!]\ \rho\ w \end{aligned}$$

Semantics of operators

$$\begin{aligned} &[\![unop]\!] : \mathbf{V} \to \mathbf{V} \\ &[\![binop]\!] : \mathbf{V} \times \mathbf{V} \to \mathbf{V} \end{aligned}$$

Semantics of dimensional operators

$$\begin{aligned} &[\![\#.d]\!]\ \rho\ w = w\ d \\ &[\![\texttt{first}.d\ E]\!]\ \rho\ w = [\![E]\!]\ \rho\ w\{d \mapsto 0\} \\ &[\![\texttt{next}.d\ E]\!]\ \rho\ w = [\![E]\!]\ \rho\ w\{d \mapsto w\ d + 1\} \\ &[\![E_1\ \texttt{@}.d\ E_2]\!]\ \rho\ w = \\ &\quad \textbf{let } n = \text{round } ([\![E_2]\!]\ \rho\ w) \\ &\quad \textbf{in } [\![E_1]\!]\ \rho\ w\{d \mapsto n\} \\ &[\![E_1\ \texttt{fby}.d\ E_2]\!]\ \rho\ w = \\ &\quad \textbf{if } w\ d = 0 \textbf{ then} \\ &\qquad [\![E_1]\!]\ \rho\ w \\ &\quad \textbf{else} \\ &\qquad [\![E_2]\!]\ \rho\ w\{d \mapsto w\ d - 1\} \\ &[\![E_1\ \texttt{asa}.d\ E_2]\!]\ \rho\ w = \\ &\quad \text{fix } (\lambda f : \mathbf{N} \to \mathbf{V}.\ \lambda n : \mathbf{N}. \\ &\qquad \textbf{if } [\![E_2]\!]\ \rho\ w\{d \mapsto n\} \textbf{ then} \\ &\qquad\quad [\![E_1]\!]\ \rho\ w\{d \mapsto n\} \\ &\qquad \textbf{else} \\ &\qquad\quad f\ (n + 1) \\ &\quad)\ 0 \\ &[\![E_1\ \texttt{wvr}.d\ E_2]\!]\ \rho\ w = \\ &\quad \text{fix } (\lambda f : \mathbf{N} \to \mathbf{N} \to \mathbf{V}.\ \lambda n : \mathbf{N}.\ \lambda m : \mathbf{N}. \\ &\qquad \textbf{if } [\![E_2]\!]\ \rho\ w\{d \mapsto n\} \textbf{ then} \\ &\qquad\quad \textbf{if } w\ d = m \textbf{ then} \\ &\qquad\qquad [\![E_1]\!]\ \rho\ w\{d \mapsto n\} \\ &\qquad\quad \textbf{else} \\ &\qquad\qquad f\ (n + 1)\ (m + 1) \\ &\qquad \textbf{else} \\ &\qquad\quad f\ (n + 1)\ m \\ &\quad)\ 0\ 0 \end{aligned}$$

Figure 3. Dynamic semantics of GLU$^{\sharp}$.

most complex ones. Notice that application of these operators implicitly requires traversing the given dimension d. This may result in non-termination. The least fixed point operator is again required in order to model this behaviour.

3.4. Dimensionality analysis

In the implementation of a multidimensional programming language, it is useful to know the maximal set of dimensions in which the value of an expression may vary. This set can be determined by a

Programs

$\mathcal{D}[\![P]\!] : \mathcal{P}(\textbf{Dim})$

$\mathcal{D}[\![D^*\ E]\!] =$
let $\delta = \text{fix}\ (\lambda\delta : \textbf{End}.\ \mathcal{D}[\![D^*]\!]\ \delta\ \delta_0)$
in $\mathcal{D}[\![E]\!]\ \delta$

Definitions

$\mathcal{D}[\![D^*]\!] : \textbf{End} \to \textbf{End} \to \textbf{End}$

$\mathcal{D}[\![\epsilon]\!]\ \delta\ \delta' = \delta'$
$\mathcal{D}[\![D\ D^*]\!]\ \delta\ \delta' = \mathcal{D}[\![D^*]\!]\ \delta\ (\mathcal{D}[\![D]\!]\ \delta\ \delta')$

$\mathcal{D}[\![D]\!] : \textbf{End} \to \textbf{End} \to \textbf{End}$

$\mathcal{D}[\![\tau\ v = E]\!]\ \delta\ \delta' = \delta'\{v \mapsto \mathcal{D}[\![E]\!]\ \delta\}$

Simple expressions

$\mathcal{D}[\![E]\!] : \textbf{End} \to \mathcal{P}(\textbf{Dim})$

$\mathcal{D}[\![v]\!]\ \delta = \delta\ v$
$\mathcal{D}[\![n]\!]\ \delta = \emptyset$
$\mathcal{D}[\![\texttt{true}]\!]\ \delta = \emptyset$
$\mathcal{D}[\![\texttt{false}]\!]\ \delta = \emptyset$
$\mathcal{D}[\![unop\ E]\!]\ \delta = \mathcal{D}[\![E]\!]\ \delta$
$\mathcal{D}[\![E_1\ binop\ E_2]\!]\ \delta = \mathcal{D}[\![E_1]\!]\ \delta \cup \mathcal{D}[\![E_2]\!]\ \delta$
$\mathcal{D}[\![\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2]\!]\ \delta =$
$\mathcal{D}[\![E]\!]\ \delta \cup \mathcal{D}[\![E_1]\!]\ \delta \cup \mathcal{D}[\![E_2]\!]\ \delta$

Expressions with dimensional operators

$\mathcal{D}[\![\#.d]\!]\ \delta = \{d\}$
$\mathcal{D}[\![\texttt{first}.d\ E]\!]\ \delta =$
$\mathcal{D}[\![E]\!]\ \delta - \{d\}$
$\mathcal{D}[\![\texttt{next}.d\ E]\!]\ \delta =$
$\mathcal{D}[\![E]\!]\ \delta$
$\mathcal{D}[\![E_1\ \texttt{fby}.d\ E_2]\!]\ \delta =$
$\mathcal{D}[\![E_1]\!]\ \delta \cup \mathcal{D}[\![E_2]\!]\ \delta \cup \{d\}$
$\mathcal{D}[\![E_1\ \texttt{@}.d\ E_2]\!]\ \delta =$
$(\mathcal{D}[\![E_1]\!]\ \delta - \{d\}) \cup \mathcal{D}[\![E_2]\!]\ \delta$
$\mathcal{D}[\![E_1\ \texttt{asa}.d\ E_2]\!]\ \delta =$
$(\mathcal{D}[\![E_1]\!]\ \delta \cup \mathcal{D}[\![E_2]\!]\ \delta) - \{d\}$
$\mathcal{D}[\![E_1\ \texttt{wvr}.d\ E_2]\!]\ \delta =$
$\mathcal{D}[\![E_1]\!]\ \delta \cup \mathcal{D}[\![E_2]\!]\ \delta$

Figure 4. Dimensionality analysis for GLU♯.

purely syntactic process, which is called *dimensionality analysis* or rank analysis [24]. Elements of the domain **End** are environments that associate GLU♯ variables with the set of dimensions in which these variables vary.

$$\delta : \textbf{End} = \textbf{Var} \to \mathcal{P}(\textbf{Dim})$$

By δ_0 we denote the element of **End** which maps all variables to the empty set. The semantic function $\mathcal{D}$, which calculates the dimensionality of GLU♯ programs, definitions and expressions, is defined in Figure 4. The most interesting cases in this definition are again those of the dimensional operators.

Let us apply dimensionality analysis to a simple GLU♯ program containing the definitions below, which have been copied from the example of Figure 6.

```
real ints  = 2 fby.x ints + 1;
real sieve = ints fby.y (sieve wvr.x sieve % prime != 0);
real prime = first.x sieve;
```

It is not necessary to understand what this program does, at this point; this will be explained in Section 6.1. Using the equations of Figure 4, we can deduce that variable `ints` varies only in dimension `x`, variable `sieve` varies in `x` and `y`, whereas variable `prime` varies only in `y`.

Dimensionality analysis is not always guaranteed to produce exact results. Consider, for example, the stream `nat` of Section 2, which produces the natural numbers in ascending order along dimension `t`. The operator `#.t` has exactly the same effect and, therefore, the expression

```
nat - #.t
```

is bound to be equal to `0` everywhere. An exact dimensionality analysis for this expression should return the empty set of dimensions. However, using the equations of Figure 4, the result that one obtains is {`t`}. Both `nat` and `#.t` vary along dimension `t` and it is not possible by a simple syntactic process to determine that their values are equal. In this sense, dimensionality analysis produces the *maximal* set of dimensions along which a multidimensional expression can vary. The exact set is guaranteed to be a subset of the maximal.

4. PROGRAMMING IN GLU♮/C++

In this section, we describe the embedding of GLU♮ into C++ from the programmer's point of view. In order to use the facilities described in this section, programmers need to include the header file `glu.hpp` in their C++ source code:

```
#include "glu.hpp"
```

Implementation details that should not concern the programmer, such as the representation of multidimensional objects, object copying and warehousing, are the subject of Section 5.

4.1. Dimensions and multidimensional objects

Programmers may declare new dimensions by creating new instances of the class `Dimension`. This instantiation does not require any parameters and new dimensions are distinct from all previously defined dimensions. For example, the following code declares a new dimension `t`:

```
Dimension t;
```

A multidimensional object is represented as an instance of the class `GLU<T>`, where `T` is the type of the object's extensions. `T` can be any C++ type that is equipped with a copy constructor (whether explicit or implicit). A multidimensional object must first be declared. For example, the following line

```
GLU<int> nat;
```

declares a multidimensional object `nat` with an extension of type `int`. Then the object's definition can be provided, e.g.

```
nat = fby(t, 0, 1 + nat);
```

which is only a little clumsier than the equivalent GLU♮ definition. Note that, for technical reasons, it is necessary for the C++ declaration of `nat` to be separated from the definition of `nat`, as this definition is recursive.

The definition of `nat` uses the C++ operators `=` and `+`, which have been overloaded and take multidimensional objects as operands. All non-dimensional operators of GLU♯ have been implemented as overloaded versions of the C++ builtin operators. However, the dimensional operators of GLU♯ have been implemented as C++ template functions, such as `fby` which is used in the definition of `nat`. Dimensions are specified in the first parameter of these functions. Operators `@`, `#` and the `if...then...else` construct of GLU♯ have been implemented as the C++ template functions `at`, `value` and `cond`, respectively.

Normal C++ objects of type `T` are automatically 'promoted' to multidimensional objects of type `GLU<T>` in the presence of operators that expect multidimensional objects. For example, in the definition of `nat`, the integers `0` and `1` are treated as constant multidimensional objects of type `GLU<int>`.

4.2. Evaluation

To evaluate the extension of a multidimensional object one needs to specify a possible world, i.e. an instance of the class `PossibleWorld`. Such an object contains a mapping of dimensions to values of type `DimValue`, which is an appropriate unsigned integer type. The constant object `PossibleWorld::empty` represents an empty mapping. Empty mappings can also be created by instantiating the class `PossibleWorld`:

```
PossibleWorld w;
```

The method `set` may be used to alter a possible world. If `t` is a dimension, the following line alters the possible world `w` by mapping the dimension `t` to value `42`.

```
w.set(t, 42);
```

To find the extension of a multidimensional object in a given possible world, one must call the object's `evaluate` method. Given the previous definitions, the following line evaluates `nat` when `t` is `42` and assigns the result, which is also `42`, to the integer variable `n`.

```
int n = nat.evaluate(w);
```

There is a distinguished object of class `PossibleWorld`, which is called `theWorld`. If the parameter to an `evaluate` method is omitted, `theWorld` is used instead. The C++ operator `<<`, which has been overloaded to handle multidimensional objects, uses `theWorld` to evaluate its operand and subsequently outputs the result. To illustrate this, the following lines write the value 42 to the standard output.

```
theWorld.set(t, 42);
cout << nat;
```

4.3. Lazy functions and methods

GLU♯ does not directly support functions, however it is possible to use the functions of C++ for this purpose. The programmer should only take into account a basic semantic difference: functions in GLU♯ are expected to be lazy (as they are in Lucid and GLU), whereas in C++ functions are eager. The kind of problems that arise because of this difference is illustrated in the following example:


```
GLU<int> f (int x)
{
   return fby(t, x, f(x) + 1);
}
```

This function is meant to produce the increasing stream of integer numbers, in dimension `t`, starting from `x`. However, this code is not correct. In C++, the occurrence of `f(x)` in the body of `f` causes the function to be called again with the same argument, leading to nontermination. In this case, it is obvious that the evaluation of `f(x)` in the body of `f` must be delayed. This is achieved by using the template `lazy`, which converts `f` to a lazy function object.

```
GLU<int> f (int x)
{
   return fby(t, x, lazy(f)(x) + 1);
}
```

In this way, it is not `f` but `lazy(f)` that is applied to `x` and our implementation causes the actual call to `f` to be delayed until the result of function `f` is evaluated.

In general, if `f` is a function that expects $\mathtt{N} \geq 0$ arguments of types `T1`, ..., `TN` and returns a result of type `GLU<R>`, then `lazy(f)` is an object of the class

```
LazyFunctionN<R, T1,..., TN>
```

This object has an overloaded function application operator, taking `N` arguments of the right types and returning a result of type `GLU<R>`.

A similar technique is used to support lazy method invocation, in case the extension of a multidimensional object is an instance of a C++ class. Let `C` be a class with a method `m` that expects $\mathtt{N} \geq 0$ arguments of types `T1`, ..., `TN` and returns a result of type `R`. Let `x` be a multidimensional object of type `GLU<C>`. Then, `lazymethod(x, C::m)` is a lazy method object of the class:

```
LazyMethod<C, R, T1,..., TN>
```

This object also has an overloaded function application operator, taking `N` arguments of the right types and returning a result of type `GLU<R>`. The extension of the result is obtained by invoking `m` on the extension of `x` and passing the supplied arguments. The reader is referred to Section 6.3 for a full example of lazy method invocation.

5. IMPLEMENTATION

Figure 5 presents the hierarchy of C++ classes and class templates in the implementation of GLU$^\natural$. Following UML notation, the hollow arrows denote the subclass relation. The class templates that are noted with a star ($*$) in this figure are repeated for each operator of C++ that will be used in embedded GLU$^\natural$ code. Those noted with two stars ($**$) are repeated for a different number of arguments, as discussed in Section 4.3.

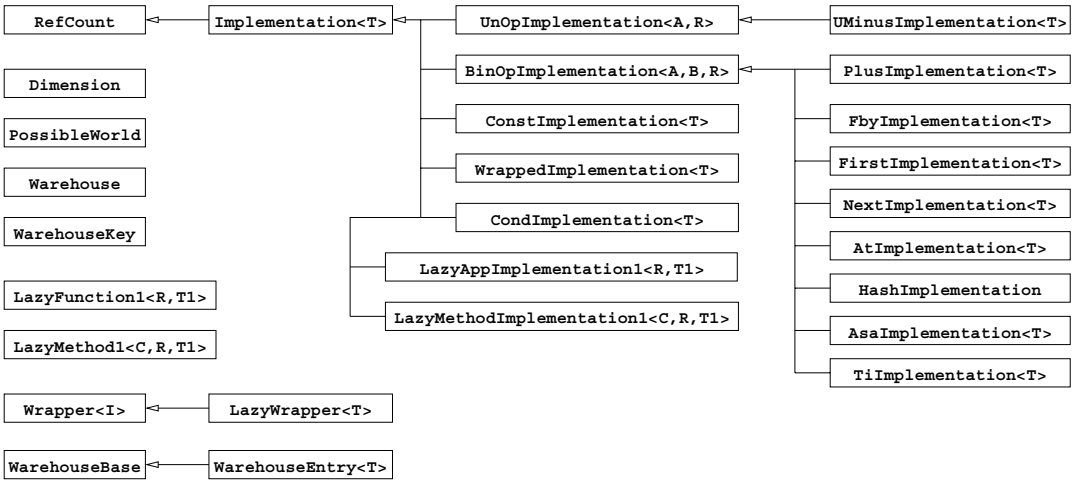

Figure 5. Class hierarchy in the implementation of GLU♮.

5.1. Wrappers and reference counting

Evaluation on demand is the driving philosophy in the implementation of a multidimensional language. For this reason, GLU♮ is implemented as an interpreter embedded in C++, with objects modelling both values and expressions that have not yet been evaluated. In contrast to C++ which supports mutable variables and objects, multidimensional objects in GLU♮ are immutable. All operations on such objects lead to the creation of new objects that, when evaluated, will force the evaluation of the operands.

A naïve embedding of GLU♮ as an interpreted language in C++ could require the frequent copying of objects, which is expensive both in terms of execution time and memory use. To avoid this problem, the implementation of GLU♮ distinguishes between *wrapper objects* and *implementation objects*, following a model known in object-oriented development as the letter/envelope idiom [25]. The objects that are frequently copied are wrappers, which contain pointers to implementations and whose copying is inexpensive. Implementations are seldom copied. In this way, one or more wrappers can refer to the same implementation and a reference counting mechanism is required. This mechanism works on the assumption that the embedded GLU♮ objects are purely functional: the programmer is not allowed to use arbitrary C++ code to alter their values.

The base classes `RefCount` and `Wrapper<I>` are used for this purpose, where `I` is the type of the wrapped implementation. When an object derived from `Wrapper<I>` is linked to an object derived from `RefCount`, the reference counter in the second object is increased. When the wrapper is destroyed or redirected to a different implementation, the reference counter is decreased; if its value becomes zero, the implementation object is destroyed.

5.2. Dimensions and possible worlds

The classes `Dimension` and `PossibleWorld` lay the foundations for the multidimensionality of GLU♮. Dimensions are simple objects which contain identifiers unique for each distinct GLU♮ dimension. Possible world objects, however, contain mappings from dimensions to unsigned integer values of type `DimValue`.

5.3. Warehouse for evaluated expressions

In order to improve the performance of C++ programs with embedded GLU♮ code, we equip our implementation with a *warehouse* of evaluated expressions. Once the value of an embedded GLU♮ expression is computed in a given possible world, this value is stored (cached) in the warehouse for future use. This is possible because of GLU♮'s purely functional character. More precisely, let us suppose that the value of a GLU♮ expression E must be computed in the possible world w. Our implementation proceeds by doing the following.

(1) It retrieves the set S of dimensions on which E depends. This set follows syntactically from the rules of dimensionality analysis given in Section 3.4. It is computed once, when the expression's implementation object is built, without significant overhead.
(2) It forms the set X of pairs (d, v), where $d \in S$ and v is the value (index) of d in w. We call X a *subworld*.
(3) It searches the warehouse for the value of E in subworld X. If this is found, nothing more needs to be done. Otherwise the implementation proceeds to the next step.
(4) It computes the value of E in w. This may require the values of other expressions in the same or in different possible worlds—in the same way, if these are found in the warehouse they need not be recomputed.
(5) After computing the value of E, it stores in the warehouse the fact that E is equal to this value in subworld X.

It should be noted here that, in contrast to other Lucid/GLU implementations that only store values of *variables* in the warehouse, our implementation stores values of arbitrary *expressions*, including their subexpressions. This leads to improved performance in the case of programs containing function calls and `wvr`. It is possible because each expression E is represented in memory by an implementation object. A unique identifier for this implementation object and the subworld X form the key of the entry stored in the warehouse that contains the value of E in X. The key is represented in our implementation by the class `WarehouseKey` and the entry by the class `WarehouseEntry<T>`. The warehouse itself is an instance of the class `Warehouse`.

A hash table is used for searching and updating the warehouse efficiently, based on the technique proposed by Faustini and Wadge [26], but significantly extended with support for multiple dimensions and dimensionality analysis, as mentioned above. The hashing function uses the identifier of E and a combination of the indices in X; when a possible match is found in the hash table, it must be determined whether X is really a subworld of w. The size of the hash table is finite and, when the warehouse is full, the newly computed values can replace older values of a lesser priority. Priority is a function of age, hits and effort required to recompute a value.

5.4. Lazy implementations and wrappers

The embedding of GLU$^\natural$ in C++ is based on a number of lazy implementation classes which represent the language constructs of GLU$^\natural$. The class `LazyWrapper<T>` is very useful: it builds wrappers for multidimensional objects, where `T` is the type of the objects' extensions. In fact, `GLU<T>` is only a shorter synonym for `LazyWrapper<T>`. Each such wrapper contains a pointer to a lazy implementation object.

The abstract class `Implementation<T>` is the base class of the hierarchy of lazy implementation objects in Figure 5. This class contains a method that evaluates a multidimensional object in a given possible world, as well as methods that calculate the set of dimensions on which such an object depends, for collaboration with the warehouse. Its subclasses correspond to the various language constructs of GLU$^\natural$, e.g. `ConstImplementation<T>` implements a constant object whose value does not vary in any dimension and `FbyImplementation<T>` implements the dimensional operator `fby`.

5.5. Implementation of `wvr`

The implementation of operator `wvr` requires special attention, in order to efficiently collaborate with the warehouse. Consider the expression `wvr(d, a, b)` and let `w` be a possible world that maps dimension `d` to m. Our implementation of `wvr` in [27] evaluates this expression in `w` by iteration: starting from point `d` $= 0$, it evaluates `b` for all points of dimension `d` until `b` becomes `true` for the $(m+1)$-th time. Using this evaluation strategy, however, even if the same expression has already been evaluated in many different possible worlds, the information stored in the warehouse does not help. To illustrate this, consider the following simple code that outputs the first $n = 1000$ even natural numbers:

```
const int n = 1000;
GLU<int> even;

even = wvr(t, nat, nat % 2 == 0);
for (int i = 0; i < n; i++)
   cout << at(t, even, i) << endl;
```

Using our implementation in [27], this code takes time quadratic in n. The problem is that each loop re-evaluates `even` from the beginning: the iteration that is required to evaluate the `wvr` always starts from 0.

To improve the performance of `wvr`, in our implementation we extend GLU$^\natural$ with a hidden dimensional operator `ti`, standing for 'truth index'. If `d` is a dimension and `E` is a Boolean multidimensional expression, then `ti.d E` is the stream of all points in dimension `d` at which `E` evaluates to `true`. Using `ti`, the operator `wvr` can be implemented as

```
a wvr.d b ≡ a @.d (ti.d b)
```

The importance of this equivalent implementation will become apparent immediately, when the evaluation strategy for the expression `ti.d b` at point `d` $= m$ is explained. An iteration is used again for this purpose. However, this time the iteration can have a better starting point if $m > 0$: it suffices to start from one plus the value of `ti.d b` at point `d` $= m - 1$. If the values of `ti.d b` are stored

```
#include <iostream.h>
#include "glu.hpp"

int main ()
{
    GLU<unsigned long int> ints, sieve, prime;
    Dimension x, y;

    ints  = fby(x, 2, ints + 1);
    sieve = fby(y, ints,
                wvr(x, sieve, sieve % prime != 0));
    prime = first(x, sieve);

    for (DimValue i=0; i<100; i++)
        cout << at(y, prime, i) << endl;
}
```

↑	⋮	⋮	⋮	⋮	⋮	
x	13	25	37	47	53	⋯
	12	23	35	43	47	⋯
	11	21	31	41	43	⋯
	10	19	29	37	41	⋯
	9	17	25	31	37	⋯
	8	15	23	29	31	⋯
	7	13	19	23	29	⋯
	6	11	17	19	23	⋯
	5	9	13	17	19	⋯
	4	7	11	13	17	⋯
	3	5	7	11	13	⋯
	2	3	5	7	11	⋯
			sieve			$y \rightarrow$

Figure 6. GLU$^\sharp$/C++ program that displays the first 100 prime numbers.

in the warehouse (as is the case with all multidimensional expressions in our implementation), there is a significant improvement in performance. The example that prints the first n even natural numbers requires only linear time.

6. EXAMPLES

The embedding of GLU$^\sharp$ in C++ leads to an interesting hybrid programming language. Its expressiveness is apparent in the examples of this section. The first two have been adapted from examples of GLU programs in the recent PhD thesis of Joey Paquet [10], whose subject was the application of multidimensional intensional programming for the coding of scientific algorithms. Only minor modifications were necessary wherever GLU's functions and `where` clauses were used. The third example gives us a taste of multidimensional object-oriented programming.

6.1. Eratosthene's sieve

The program of Figure 6 computes the first 100 prime numbers in increasing order, using the algorithm known as *Eratosthene's sieve*. The prime numbers are computed in the infinite stream `prime`, along the `y` dimension. Let us notice that the implementation of this algorithm in GLU$^\sharp$ requires just three lines of code, the first of which defines the stream `ints` of natural numbers greater than or equal to 2, along dimension `x`.

```
real ints  = 2 fby.x ints + 1;
real sieve = ints fby.y (sieve wvr.x sieve % prime != 0);
real prime = first.x sieve;
```

The variable `sieve` is a two-dimensional stream varying in `x` and `y`; we can think of its values in terms of rows and columns as shown in Figure 6. The stream `prime` is defined as the first row of


```
#include <iostream.h>
#include "glu.hpp"

int main ()
{
   GLU<int> hamming;
   Dimension t;

   hamming =
      fby(t, 1,
         merge(t,
            merge(t,
               2 * hamming,
               3 * hamming),
            5 * hamming));

    for (DimValue i=0; i<100; i++)
        cout << at(t, hamming, i)
             << endl;
}
```

```
template <class T>
GLU<T> upon (const Dimension & d,
             const GLU<T> & x,
             const GLU<bool> & y)
{
    GLU<DimValue> w;

    w = fby(d, 0, cond(y, w+1, w));
    return at(d, x, w);
}

template <class T>
GLU<T> merge (const Dimension & d,
              const GLU<T> & x,
              const GLU<T> & y)
{
    GLU<T> xx, yy;

    xx = upon(d, x, xx <= yy);
    yy = upon(d, y, yy <= xx);
    return cond(xx < yy, xx, yy);
}
```

Figure 7. GLU$^{\natural}$/C++ program that displays the first 100 Hamming numbers.

`sieve`, i.e. when `x` is 0. The first column of `sieve` contains `ints`. Each subsequent column contains the elements of the previous column that are not multiples of the first element. This is achieved with the operator `wvr`, which selects the elements of `sieve`, along dimension `x`, for which `sieve % prime != 0` is true.

6.2. Hamming numbers

The natural numbers of the form $2^i \cdot 3^j \cdot 5^k$, where $i, j, k \in \mathbb{N}$, are called *Hamming numbers*. The following GLU (not GLU$^{\natural}$) program computes all Hamming numbers in increasing order in the infinite stream `hamming` along dimension `t`. The function `merge` takes as input two streams sorted in increasing order along dimension `d` and produces a single stream, sorted in increasing order with duplicate elements removed. The dimensional operator `upon.d` only allows its left operand to advance in dimension `d` if its right operand is true. (For a longer description of how this program works, the reader is referred to [10].)

```
merge.d (x, y) = if xx < yy then xx else yy fi
   where xx = x upon.d (yy >= xx);
         yy = y upon.d (xx >= yy);
   end;

dimension t;
hamming = 1 fby.t merge.t(merge.t(2 * hamming, 3 * hamming), 5 * hamming);
```

```
#include <iostream.h>
#include "glu.hpp"

typedef enum {
   GREEN  = 0,
   YELLOW = 1,
   RED    = 2
} State;

const int COLORS = 3;
const int timePerLight [COLORS] =
   { 5, 1, 8 };
const char * const stateName [COLORS] =
   { "green", "yellow", "red" };

class TrafficLight
{
private:
   State state;
   int   timer;

public:
   TrafficLight (State s, int t = 0) :
      state(s)
   {
      if (t > 0)
         timer = t;
      else
         timer = timePerLight[s];
   }
   TrafficLight next () const
   {
      int t = timer - 1;

      if (t <= 0) {
         State s = (state + 1) % COLORS;
         t = timePerLight[s];
         return TrafficLight(s, t);
      }
      else
         return TrafficLight(state, t);
   }

   friend ostream & operator << (
      ostream & str,
      const TrafficLight & light)
   {
      str << stateName[light.state];
      return str;
   }
};

int main ()
{
   GLU<TrafficLight> light;
   Dimension t;

   light = fby(t, TrafficLight(RED),
               lazymethod(light,
                  TrafficLight::next)());

   for (DimValue i=0; i<100; i++)
      cout << at(t, light, i) << endl;
}
```

Figure 8. GLU$^\natural$/C++ program that simulates a traffic light.

We should note here that this program uses features of GLU that are not supported in GLU$^\natural$: functions, the `where` structure for the definition of local variables and the operator `upon`. Fortunately, the first two can be expressed in C++ and `upon` can be expressed in terms of the other dimensional operators. The C++ program in Figure 7 is an almost direct translation of the above. The definition of `upon` can be found in introductory texts of GLU, e.g. [6]. The implementations of both `upon` and `merge` illustrate how it is possible to define polymorphic functions (templates) that have dimensions as parameters. Notice, however, that `upon` and `merge` are eager C++ function templates. They can be defined in this way because they are not recursive. In Section 7 it will be verified that this has a huge impact on the program's performance.

6.3. Traffic lights

In this example we show how arbitrary C++ classes can be used in GLU♮/C++ programs as the extensions of multidimensional values. The program shown in Figure 8 simulates the behaviour of a traffic light over a dimension `t`, which represents time. At a given moment, the light is represented by an object of the class `TrafficLight`. The field `state` contains the current colour of the light, whereas `timer` contains the time that remains until the light changes colour. Method `next` informs us of what will happen at the next moment in time. A light stays green for 5 moments (units of time), yellow for 1 moment and red for 8 moments. The cycle is then repeated.

The most interesting part of this program is the definition of variable `light`:

```
light = fby(t, TrafficLight(RED), lazymethod(light, TrafficLight::next)());
```

In simple words, this says that the light is red when `t` is 0 and each subsequent value is the result of applying method `next` to the current value. One would simply expect `light.next()` in the second part of the `fby`. However, this is not possible because `next` is a method of `TrafficLight`, not `GLU<TrafficLight>`, and the dot operator cannot be overloaded in C++. As mentioned in Section 4.3, the function template `lazymethod` delays the invocation of method `TrafficLight::next` in such a way that the method is applied to the extensions of `light` when they are evaluated.

7. EVALUATION

Table I shows timings in seconds for the first two examples of Section 6 on a Sun Ultra workstation (Ultrasparc I, 140 MHz, 192 MB RAM). For each example, the first column shows the size of the problem, i.e. which element of `prime` or `hamming` is to be computed, and the next columns show the execution time. We compare our implementation with: (i) the GLU implementation [28], v. 960730; (ii) the FC++ library supporting functional programming in C++ [29], v. 1.1 beta; and (iii) the Hugs Haskell Interpreter, v. January 2001 beta. The GNU C/C++ compiler, v. 2.95.2, was used whenever a C/C++ compiler was required, with the best possible optimization flags. Missing timings in Table I either mean that the execution time would be extremely long, or that the corresponding program did not terminate successfully.

We should remark however that this comparison is far from fair. Of all implementations, only GLU♮/C++ and GLU support multiple dimensions and could be compared with more or less equivalent programs. In the case of FC++ and Hugs we had to use lazy infinite lists instead of one-dimensional streams and we circumvented the use of two-dimensional streams in the first example. In the second example we give two timings for GLU and Hugs. Column (1) corresponds to a translation of the source program that is as faithful to the original as possible. By contrast, the program in column (2) uses the builtin operator `upon` in the case of GLU and a custom `merge` function in the case of Hugs, that differs from the one in Figure 7. The same `merge` function is used in FC++. In all cases, the performance is significantly improved.

In the case of prime numbers, it is obvious that true multidimensional implementations perform poorly compared with lazy lists. GLU♮/C++ takes approximately the same time as GLU. This result is highly in favour of our implementation; it would be reasonable for the GLU compiler to do better, since

Table I. Performance results.

Prime numbers					Hamming numbers						
							GLU			Hugs	
n	GLU♮/C++	GLU	FC++	Hugs	n	GLU♮/C++	(1)	(2)	FC++	(1)	(2)
10	<0.1	<0.1	<0.1	1.1	10	<0.1	0.5	0.4	<0.1	1.3	1.3
50	0.6	0.7	<0.1	1.2	20	<0.1	7.2	6.0	<0.1	1.3	1.3
100	2.5	2.5	0.2	1.5	50	0.1	1456.5	167.3	0.3	1.6	1.3
200	10.0	9.8	0.9	2.5	100	0.2			2.2	2.7	1.3
300	22.5	24.6	2.0	4.3	200	0.4				7.3	1.3
400		39.4	3.6	6.5	500	0.9				41.7	1.4
500		60.4	5.6	9.5	1000	1.7					1.5
1000		243.0		33.7	2000	3.4					1.8

it translates its source program to C code which is subsequently compiled and executed. In the case of Hamming numbers, the results are quite surprising: for GLU♮/C++ and the custom Hugs program execution time increases linearly in n, whereas for other programs execution time increases much more rapidly. This implies that the calls to the `merge` function introduce unnecessary recalculations by confusing GLU's warehouse and the lazy call-by-need evaluation strategy. In the GLU♮/C++ program, `upon` and `merge` are eager C++ function templates; this allows intermediate results to be stored in the warehouse and avoids unnecessary recalculations. If these functions are defined using `lazy`, this advantage is lost and the program performs slightly better than the unoptimized GLU program.

Overall, we find our performance results satisfactory and encouraging, taking into account that GLU♮/C++ is a hybrid system implemented as an interpreter embedded in C++.

8. CONCLUSION

Several attempts have been made to embed support for the lazy functional programming paradigm in object-oriented languages with notable success. As far as we know, the integration of multidimensional and object-oriented programming in a single language has inspired two different approaches. The first sought the extension of Lucid with objects [30] and culminated with *OLucid*, a Java-like language encompassing much of first-order Lucid with emphasis on multithreaded and distributed execution [31]. In a symmetric way, the second approach sought the extension of existing object-oriented languages with multidimensional features [18].

Our work presented in this paper is aligned with the second approach. It accomplishes a clean embedding of a small multidimensional core (GLU♮) in a mainstream object-oriented programming language (C++). Our contribution with respect to previous work is twofold. First, more multidimensionality can be supported without changing in the least the syntax and semantics of the host language. GLU♮ supports most of the dimensional operators of GLU and, by using features of C++, the embedded language supports (uncurried) multidimensional higher-order functions (both eager

and lazy), dimensional abstraction, local definitions and parametric polymorphism. Second, the use of dimensionality analysis has improved the overall performance of our implementation.

GLU♯/C++ combines the expressive power of C++ with a small and purely functional multidimensional core. In this way, it exploits the expressiveness of the multidimensional programming model, while at the same time staying compatible with the host language C++. The evaluation of our implementation is not yet complete. The embedding that we have achieved so far is quite natural, however few syntactic improvements can be expected in this direction. We believe that performance can be significantly improved, especially with a better design of the warehouse.

One interesting direction for future research is the investigation of the hybrid language's applications. As GLU♯/C++ can support multidimensional variables whose extensions are elaborate data structures, it remains to be seen whether multidimensional programming can find applications in areas of computation other than scientific computing [10]. Of course, the data structures must be purely functional, but it has been shown that this does not necessarily imply poor performance [32]. Another direction worth investigating is the introduction of branching dimensions in GLU♯. Apart from the possible applications that this entails, the existence of branching dimensions would allow us to compare our implementation of lazy functions with the standard implementation of functions in intensional languages.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees, whose constructive comments and suggestions were essential for improving the presentation and the paper's structure. An earlier, slightly extended version of this paper in Greek is available as a technical report [27].

REFERENCES

1. Dowty DR, Wall RE, Peters S. *Introduction to Montague Semantics*. Kluwer: Dordrecht, 1981.
2. Ashcroft EA, Wadge WW. Lucid, a nonprocedural language with iteration. *Communications of the ACM* 1977; **20**:519–526.
3. Wadge WW, Ashcroft EA. *Lucid, the Dataflow Programming Language*. Academic Press: New York, 1985.
4. Du W. An intensional approach to parallel programming. *IEEE Parallel and Distributed Technology: Systems and Applications* 1993; **1**:22–32.
5. Ashcroft EA, Faustini AA, Jagannathan R. An intensional language for parallel applications programming. *Parallel Functional Languages and Compilers*, Szymanski BK (ed.). ACM Press: New York, 1991; 11–49.
6. Ashcroft EA, Faustini AA, Jagannathan R, Wadge WW. *Multidimensional Programming*. Oxford University Press: Oxford, 1995.
7. Faustini AA, Lewis EB. Toward a real-time dataflow language. *Tutorial: Hard Real-Time Systems*, Stankovic JA, Ramamritham K (eds.). IEEE Computer Society Press: Los Alamitos, CA, 1989; 139–145.
8. Plaice J, Khédri R, Lalement R. From abstract time to real time. *Proceedings of the 6th International Symposium on Lucid and Intensional Programming*. Université Laval: Québec City, Canada, 1993; 83–83.
9. Rao P, Jagannathan R. Developing scientific applications in GLU. *Proceedings of the 7th International Symposium on Lucid and Intensional Programming*. SRI International: Menlo Park, CA, 1994; 45–52.
10. Paquet J. Intensional scientific programming. *PhD Thesis*, Département d'Informatique, Université Laval, Québec, April 1999.
11. Paquet J, Plaice J. On the design of an indexical query language. *Proceedings of the 7th International Symposium on Lucid and Intensional Programming*. SRI International: Menlo Park, CA, 1994; 28–36.
12. Du W, Wadge WW. A 3D spreadsheet based on intensional logic. *IEEE Software* 1990; **7**:78–89.
13. Agi I. GLU for multidimensional signal processing. *Intensional Programming I*, Orgun MA, Ashcroft EA (eds.). World Scientific: Singapore, 1996; 135–148.

14. Tao S. Indexical attribute grammars. *PhD Thesis*, Department of Computer Science, University of Victoria, 1994.
15. Plaice J, Wadge WW. A new approach to version control. *IEEE Transactions on Software Engineering* 1993; **19**:268–276.
16. Yildirim T. Intensional HTML. *Master's Thesis*, Department of Computer Science, University of Victoria, 1997.
17. Hudak P. Modular domain specific languages and tools. *Proceedings 5th International Conference on Software Reuse*, June 1998, Devanbu P, Poulin J (eds.). IEEE Computer Society Press: Los Alamitos, CA, 1998; 134–142.
18. Rondogiannis P. Adding multidimensionality to procedural programming languages. *Software Practice and Experience* 1999; **29**:1201–1221.
19. Rondogiannis P, Gergatsoulis M, Panayiotopoulos T. Branching-time logic programming: The language Cactus and its applications. *Computer Languages* 1998; **24**(3):155–178.
20. Rondogiannis P, Wadge WW. First-order functional languages and intensional logic. *Journal of Functional Programming* 1997; **7**:73–101.
21. Rondogiannis P, Wadge WW. Higher-order functional languages and intensional logic. *Journal of Functional Programming* 1999; **9**:527–564.
22. Stoy JE. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press: Cambridge, MA, 1977.
23. Tennent RD. *Semantics of Programming Languages*. Prentice-Hall: Englewood Cliffs, NJ, 1991.
24. Dodd C. Rank analysis in the GLU compiler. *Intensional Programming I*, Orgun MA, Ashcroft EA (eds.). World Scientific: Singapore, 1996; 76–82.
25. Coplien JO. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley: Reading, MA, 1992.
26. Faustini A, Wadge WW. An eductive interpreter for the language pLucid. *SIGPLAN Notices (Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques)* 1987; **22**(7):86–91.
27. Papaspyrou NS, Kassios IT. GLU$^\sharp$/C++: Theory and implementation. *Technical Report CSD-SW-TR-1-00*, Department of Electrical and Computer Engineering, Software Engineering Laboratory, National Technical University of Athens, 2000 (in Greek).
28. Jagannathan R, Dodd C. GLU programmer's guide, version 1.0. *Technical Report 96-06*, SRI International, Computer Science Laboratory, 1996.
29. McNamara B, Smaragdakis Y. Functional programming in C++. *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, September 2000. ACM Press: New York, 2000; 118–129.
30. Freeman-Benson B. Lobjcid: Objects in Lucid. *Proceedings of the 1991 Symposium on Lucid and Intensional Programming*, April 1991. SRI International: Menlo Park, CA, 1991; 80–87.
31. Zhao Q. Implementation of an object-oriented intensional programming system. *Master's Thesis*, University of New Brunswick, Canada, September 1997.
32. Okasaki C. *Purely Functional Data Structures*. Cambridge University Press: Cambridge, 1998.

