The CUBE Language *

Marc A. Najork Simon M. Kaplan

Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

CUBE is a three-dimensional, visual, statically typed, higher-order logic programming language. CUBE will eventually be embedded into a virtual-reality-based programming environment that allows a user to manipulate a CUBE program simply by grabbing, placing, and moving its components.

1 Introduction

CUBE is a new programming language which combines several innovative features, namely

- a visual, three-dimensional syntax, which shall eventually make it possible to edit CUBE programs in a virtual-reality based programming environment,
- a static, polymorphic type system, as it is used by many functional languages
- a Horn-Logic based semantics which is higher-order in that it treats predicates as first class values

As the semantics of Horn-Logic is inherently parallel, so is the semantics of CUBE. We hope to come up with a parallel implementation of CUBE.

The visual aspect of CUBE has been strongly influenced by Kimura's Show-and-Tell [9] and by work on a type system for Show-and-Tell [15]. Much work has been done on polymorphic type systems for functional languages [5], and, more recently, for logic languages [13][17], which applies almost directly to our work. A lot of work has been done as well on languages based on (first-order) Horn-Logic [3][10], and some on higher-order variants thereof [11]. Finally, much attention has been devoted recently to parallel implementations of logic-based languages [4][15].

2 An example program

The program shown in Fig. 1 gives the flavor of a CUBE program. It shows the recursive definition of the factorial predicate. This predicate is represented by a predicate definition cube with two ports, an input and an output port. Values enter the cube through the port on the left side and get distributed over the two inner planes. Planes are evaluated independently and concurrently.

If the input value *i* is not 0, and thus does not unify with the value 0 inside the lower left holder, the lower plane fails, otherwise, the unification succeeds, and, as there are no other conditions to satisfy, the value 1 inside the lower right holder cube will "flow" to the output port (be unified with it), and thus will be returned as a result.

The value *i* of the input port will also be distributed to the upper plane. Here it will flow into the predicate *greater*, which also receives the value 0 from the holder cube to its right as a second argument. The predicate succeeds if $i > 0$, otherwise, it fails and thereby causes the entire upper plane to fail. *i* also flows into the *minus* predicate, together with the value 1 from the holder cube right of *minus*. The result of the subtraction flows into a recursive occurrence of the factorial cube. Its result flows in turn into a multiplication cube, which also takes *i*, and returns the result to the right port.

In the next chapter, we will present the syntax and the semantics of CUBE.

3 The Language

In the following section, we will define the CUBE language in an informal fashion. In order to give readers who are familiar with traditional textual functional and logic programming languages a better understanding of the semantics of our language, we will also define a textual version of CUBE, which is similar to many functional and logic languages, and point out the correspondence between visual CUBE and textual CUBE.

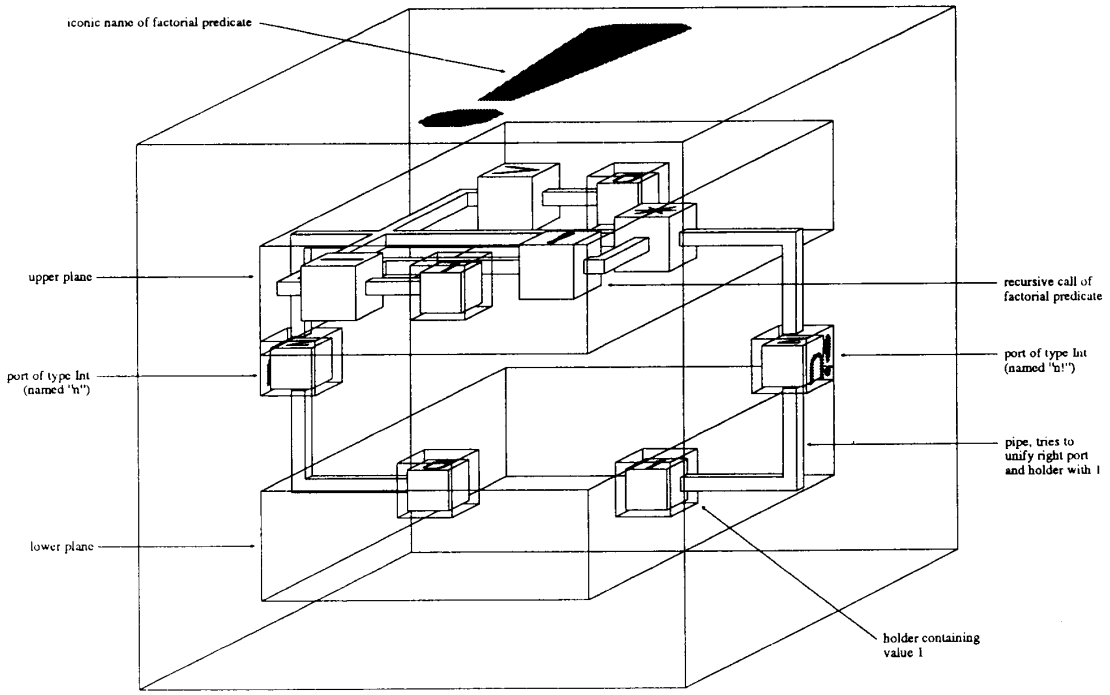
The basic syntactic elements of CUBE are *cubes*, *planes*, *pipes*, and *icons*. We can categorize cubes into transparent *holder cubes* (or *holders* for short), transparent *abstraction cubes*, and opaque *reference cubes*. We can also categorize cubes into grey *type cubes*, and blue *value cubes* (we omit a discussion on the role of colors in execution animation). *Type cubes* hold, define, or refer to types, *value cubes* hold, define, or refer to values. Values are constructors, predicates, and variable instances.

A *holder cube* either holds or will eventually hold a value (or type). Two (or more) holder cubes may be connected by a pipe, if the types of the values (or types) inside them are compatible (where the type of a type is TYPE). If two holder cubes are connected, then the values (or types) they contain are unified. If a holder cube does not hold a value yet, CUBE instead fills it with the cube representing the inferred type of the value-to-be.

A *naming cube* is a special kind of holder cube, which carries an icon on its top. It defines a name for the value (or type) inside it. The scope of this name is the inside of the enclosing definition cube. Within this scope, each occurrence of a *reference cube* carrying the same icon will refer to the value (or type) inside the naming cube.

So, the two principal mechanisms in CUBE to propagate values (and types) are pipes, which are direct and explicit point-to-point links, and for which the issue of scope does not arise, and iconic names, which "broadcast" values from a naming cube to all corresponding reference cubes within a given scope.

* This work was supporeted by the National Science Foundation under grant CCR-9007195



$fact = \lambda\{in{:}Int,out{:}Int\}.(in{=}0 \wedge out{=}1) \vee$
$(greater\ \{arg_1{=}in,arg_2{=}0\} \wedge minus\ \{arg_1{=}in,arg_2{=}1,res{=}y_1\} \wedge fact\ \{in{=}y_1,out{=}y_2\} \wedge times\ \{arg_1{=}in,arg_2{=}y_2,res{=}out\})$

Fig. 1 : Definition of Factorial

There are two kinds of *abstraction cubes*: *type abstraction cubes*, and *predicate abstraction cubes* (predicates are, as we will see, a special kind of value). Abstraction cubes must be surrounded (and thereby named) by a naming cube. For convenience, we usually show the two cubes as one, and refer to them as a *definition cube*. Definition cubes define "functions" which take a number of arguments, and return a result. In traditional textual languages, arguments are bound to formal parameters by position: The first argument is bound to the first parameter, etc. In CUBE, however, binding is done by name: Formal parameters are represented by naming cubes (called *ports*) inside the definition cube, such that their icons touch its walls. A reference cube corresponding to the named definition cube will have the transparent ports set into its opaque body. These ports can be moved over the surface of the reference cube, and are identified by their icon. They can be filled or supplied through a pipe with values, i.e. arguments. Inside the definition cube, one can either use pipes to propagate the values of ports, or use reference cubes to refer to them.

Note that we somewhat mix two concepts here: given a definition cube *A* with port *B*, we use *B*'s icon to refer to *B* within the body of *A*. But, given a reference to *A*, we use *B*'s icon to *identify* a port (we could not *refer* to this port, though, as *B* is local to the scope of *A*). So, we use internal names as external labels. This mixing of concepts, however, reduces the visual complexity of programs.

A definition cube contains vertically stacked boxes, called *planes*. In CUBE, vertical extent indicates alternatives: in the context of types, sum types, in the context of logic, disjunction. Horizontal extent indicates combination: Product types in the context of types, and conjunction in the context of logic.

3.1 Types

In the following, we assume the reader to be familiar with the Hindley-Milner type system [5], which is used in many functional languages, like ML [12]. The fundamental ideas of this type system have also been applied to logic languages, namely Prolog [13][16]. Table 1 shows the textual syntax of types and type definitions (which is identical to the standard format, except that arguments are associated with parameters by name rather than by position). A *type definition tdf* of the form

$K = \Lambda\{X_1{:}\mathrm{TYPE},\ldots,X_m{:}\mathrm{TYPE}\}.V_1 + \ldots + V_n \quad (m{\geq}0,n{>}0)$

is represented by a type definition cube (i.e. a type abstraction cube, representing the Λ, together with a type naming cube with icon *K*, naming a type constructor), and ports with names $X_1,\ldots,X_k$. These ports take types, so their arguments are of type TYPE. TYPE is represented as an opaque grey cube

K	$\in$ TypeConstructor $\subseteq$ TypeVariable
k	$\in$ Constructor $\subseteq$ NamedVariable
p	$\in$ PredicateName $\subseteq$ NamedVariable
X	$\in$ TypeVariable
x	$\in$ NamedVariable $-$ $\{icon_1, icon_2, \ldots\}$
y	$\in$ UnnamedVariable $-$ $\{y_1, y_2, \ldots\}$
z	$\in$ Variable $-$ NamedVariable $\cup$ UnnamedVariable
α_i	$\in$ UninstantiatedTypeVariable
a_i	$\in$ UninstantiatedVariable

Types

tdf	$:: K - \Lambda\{X_1{:}\text{TYPE},\ldots,X_m{:}\text{TYPE}\}.V_1 + \ldots + V_n$	$(m\geq 0, n>0)$
V	$:: k\ \{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\}$	$(n\geq 0)$
σ	$:: X_i \mid K\ \{X_1{-}\sigma_1,\ldots,X_n{-}\sigma_n\}$	$(n\geq 0)$
τ	$:: \alpha_i \mid K\ \{X_1{-}\tau_1,\ldots,X_n{-}\tau_n\} \mid \{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\} \rightarrow \tau_0$	$(n\geq 0)$

Values

prg	$:: con$	
df	$:: pdf \mid tdf$	
pdf	$:: p - \lambda\{x_1{:}\tau_1,\ldots,x_k{:}\tau_k\}.df_1 \wedge \ldots \wedge df_m \wedge (con_1 \vee \ldots \vee con_n)$	$(k\geq 0, m\geq 0, n\geq 0)$
con	$:: df_1 \wedge \ldots \wedge df_m \wedge af_1 \wedge \ldots \wedge af_n$	$(m\geq 0, n\geq 0)$
af	$:: z - t \mid z\ \{x_1{-}t_1,\ldots,x_n{-}t_n\}$	$(n\geq 0)$
t	$:: z\ \{x_1{-}t_1,\ldots,x_n{-}t_n\} \mid t\ \{x_1'{\leftarrow}x_1,\ldots,x_n'{\leftarrow}x_n\}$	$(n\geq 0)$

The relationship between pipes, reference cubes, naming cubes, holders, and textual variables is as follows: Given a (visual) CUBE program, assign a variable $icon_i$ to each naming and each reference cube. To each pipe or unnamed holder connected (directly or indirectly) to the port $icon_j$ of a definition cube, assign the same variable. Assign a variable y_i to each pipe and holder cube not labelled yet, such that (directly and indirectly) connected pipes and holders have the same name, and unconnected pipes and holders have different names.

Table 1 : A textual syntax for CUBE

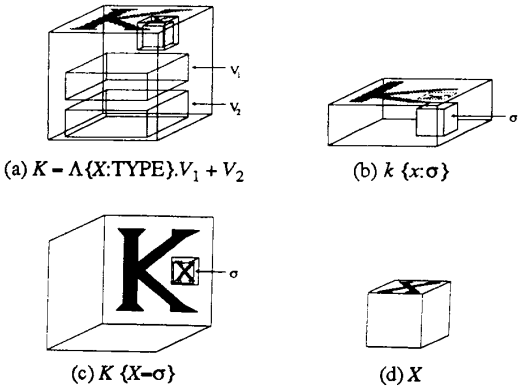
without any icons. By convention, we place ports for type parameters on the top of the definition cube. Inside the type definition cube are m different planes $V_1,\ldots,V_m$, which represent the m different variants of the sum type. Fig. 2.a shows a type definition.

A *variant* V has the form $k\ \{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\}$ $(n\geq 0)$. V is represented by a plane with a transparent icon k on top, naming the constructor. Inside the plane are m opaque type cubes $\sigma_1,\ldots,\sigma_n$. Above each type cube σ_i is a transparent icon x_i, which serves as a parameter name for the constructor (see below). Fig. 2.b shows a variant.

A type σ can have two forms:

(a) $K\ \{X_1{-}\sigma_1,\ldots,X_n{-}\sigma_n\}$ $(n\geq 0)$, where K is a type constructor defined by a type definition cube within the current scope. The type is represented by a type reference cube with icon K and ports named $X_1,\ldots,X_n$, and filled with types $\sigma_1,\ldots,\sigma_n$. Fig. 2.c shows this form.

(b) X, which can be represented either by a type reference cube referring to a port (a type parameter bound in the Λ-abstraction), or by a type holder cube connected (directly or indirectly) to a port by a pipe. Fig. 2.d shows this form.



(a) $K - \Lambda\{X{:}\text{TYPE}\}.V_1 + V_2$ (b) $k\ \{x{:}\sigma\}$

(c) $K\ \{X{-}\sigma\}$ (d) X

Fig. 2 : Visual syntax of type definitions

In general, the user never has to declare the type of an expression, types can be inferred automatically. Type inference algorithms are described in [5] [16].

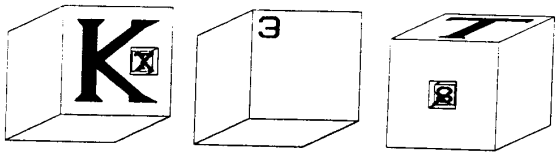
If a holder cube does not hold a value yet, CUBE will instead show the inferred type of the value-to-be inside the holder. A *type expression* τ can have three forms:

(a) A *type constructor application* $K\ \{X_1{-}\tau_1,\ldots,X_n{-}\tau_n\}$ $(n\geq 0)$, where K is a type constructor defined by a type definition cube within the current scope. The type expression is represented by a type reference cube with icon K and ports named $X_1,\ldots,X_n$, and filled with type expressions $\tau_1,\ldots,\tau_n$.

(b) an *uninstantiated type variable* α_i, represented by an opaque grey cube (the representation of TYPE) with the grey number i in its top left corner.

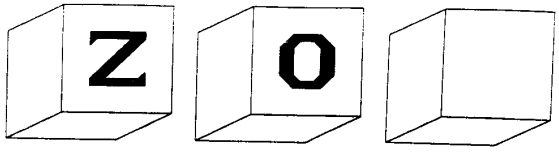
(c) a *function type* $\{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\} \rightarrow \tau_0$, represented by the type cube representing τ_0, with n ports named $x_1,\ldots,x_n$ and filled with types $\tau_1,\ldots,\tau_n$ set into its side.

Fig. 3 shows the different forms of type expressions.



(a) $K\ \{X{-}T\}$ (b) α_3 (c) $\{x{:}S\} \rightarrow T$

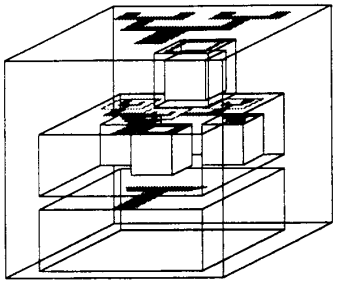
Fig. 3 : Visual syntax of type expressions



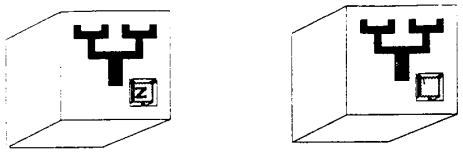
(a) *Int* (b) *o* (c) TYPE

Fig. 4 : Some predefined types

Fig. 4 shows the representation of some predefined types. Fig. 5 shows the definition of a polymorphic tree type, and instances of it.



Tree=Λ{*X*:*TYPE*}.
tree{*root*:*X*,*left*:*Tree* {*X*=*X*},*right*:*Tree* {*X*=*X*}} + *emptytree*

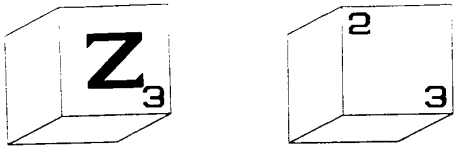


Tree {*X*=*Int*} *Tree* {*X*=α_1}

Fig. 5 : Tree Type

3.2 Uninstantiated Variables

In Horn Logic variables can be either bound to values, or they can be left uninstantiated. We treat uninstantiated variables as first-class-values. An *uninstantiated variable* a_i of type τ is represented by the grey opaque type cube representing τ, with the blue number i in its bottom right corner. Fig. 6 shows two uninstantiated variables.



a_3:*Int* a_3:α_2

Fig. 6 : Uninstantiated Variables

3.3 Constructors

Type definitions give rise to *constructors*. Constructors are used to construct values, and are first-class-values themselves. A type definition

$K = \Lambda\{X_1\text{:TYPE},...,X_m\text{:TYPE}\}. \ldots + k\ \{x_1{:}\sigma_1,...,x_n{:}\sigma_n\} + \ldots$

gives rise to a constructor k of type

$\{x_1{:}\tau_1,...,x_n{:}\tau_n\} \rightarrow K\ \{X_1{=}\alpha_1,...,X_m{=}\alpha_m\}$

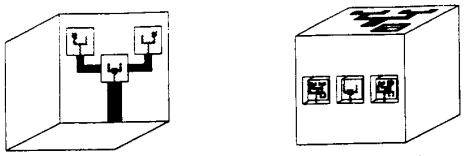
where $\alpha_1,...,\alpha_m$ are new type variables and $\tau_i = \sigma_i[\alpha_j/X_j]$ for all i,j.

A constructor can be applied to values by filling (some of) its ports with values or by supplying them through pipes. An application $k\ \{x_1'{=}t_1',...,x_m'{=}t_m'\}$ of terms $t_1',...,t_m'$ of types $\tau_1',...,\tau_m'$ to a constructor k is well-typed if k is of type $\{x_1{:}\tau_1,...,x_n{:}\tau_n\} \rightarrow \tau_0$ ($n \geq m$), each x_i' corresponds to an x_j, and τ_i' and τ_j are compatible types, yielding a most general type unifier Θ. The type τ of the result is

$\{x_j{:}\tau_j\Theta \mid x_j$ does not match any $x_i'\} \rightarrow \tau_0\Theta$

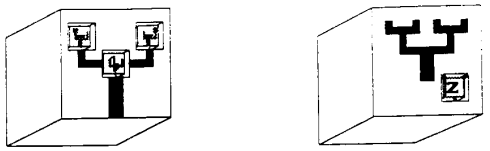
or, if $\tau = \{\} \rightarrow \tau_0\Theta$, $\tau = \tau_0\Theta$.

Fig. 7 shows the *tree* constructor and its type, Fig. 8 the value *tree* {*root*=1,*left*=*emptytree*,*right*=*emptytree*} and its type. 1 is a nullary constructor belonging to the type *Int*. Fig. 9 shows a curried application of a constructor.



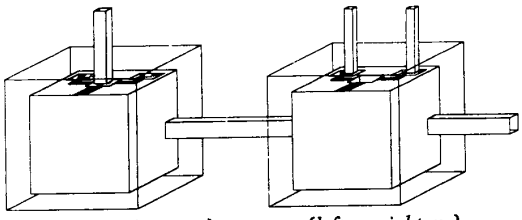
tree {*root*:α_1,*left*:*Tree* {*X*=α_1},right: *Tree* {*X*=α_1}} → *Tree* {*X*=α_1}

Fig. 7 : *tree* and its type



tree {*root*=1,*left*=*emptytree*,*right*=*emptytree*} *Tree* {*X*=*Int*}

Fig. 8 : A tree value and its type



y_2 = *tree* {*root*=y_1} ∧ y_5 = y_2 {*left*=y_3,*right*=y_4}

Fig. 9 : Curried constructor application

3.4 Predicates

In the following, we assume the reader to be familiar with Horn Logic [10] and Prolog [3]. Formally, an n-ary predicate is an n-ary relation, i.e. a set of n-tuples. For a given n-tuple x, the predicate P is said to hold if $x \in P$. For our purposes, it is more convenient to think of P as an n-ary function, which, when applied to x, returns either *success* or *failure*, depending on whether P holds for x or not. *success* and *failure* belong to the predefined type o (see Fig. 4.b), i.e. o = *success* + *failure*. So the type of an n-ary predicate P which takes arguments of type $\tau_1,...,\tau_n$ at parameters $x_1,...,x_n$ is $\{x_1{:}\tau_1,...,x_n{:}\tau_n\} \rightarrow o$. The type rule for applying constructors to values holds as well when applying predicates to values (with τ_0 being o).

Let us establish a relation between CUBE and Prolog. The Prolog n-ary predicate p defined by

$p(t_{11},...,t_{1n})$:- $g_{11},...,g_{1m_1}$.
...
$p(t_{k1},...,t_{kn})$:- $g_{k1},...,g_{km_k}$.

can be normalized to the (Prolog) predicate

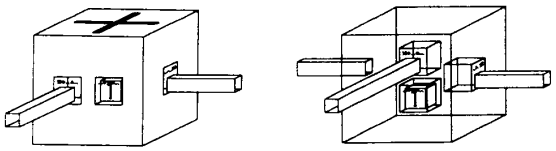
$p(x_1,...,x_n)$:- $(x_1=t_{11},...,x_n=t_{1n},g_{11},...,g_{1m_1})$;...;
$(x_1=t_{k1},...,x_n=t_{kn},g_{k1},...,g_{km_k})$.

In CUBE, we λ-abstract the predicate to

$p = \lambda x_1,...x_n.$
$(x_1=t_{11} \wedge ... \wedge x_n=t_{1n} \wedge g_{11} \wedge ... \wedge g_{1m_1}) \vee ... \vee$
$(x_1=t_{k1} \wedge ... \wedge x_n=t_{kn} \wedge g_{k1} \wedge ... \wedge g_{km_k})$

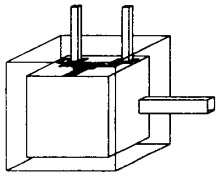
We also use binding-by-name instead of binding-by-position, and we infer types for parameters. In addition, we allow for *local type and predicate definitions* within a predicate (i.e. we introduce *nesting*). Finally, we treat predicates as first-class values, meaning that they can be passed around through pipes, and supplied as arguments to constructors and predicates. In this sense, CUBE is a *higher-order Horn Logic* language. Note, however, that CUBE uses only first-order unification (higher-order unification has been shown to be undecidable [8]). Unifying two predicates does not mean to test if they describe the same relation (which is undecidable), but just to test whether they *syntactically* unify (which is simple structural unification). There are other, more powerful — and more expensive — higher-order variants of Horn Logic (cf. [11]).

Given two holder cubes connected by a pipe, such that one is inside a plane and the other one is outside, we refer to the cube inside the plane as to the *inner* cube. If two holder cubes are not separated by a plane, we say they are on the *same level.*



plus $\{arg_1=y_1,arg_2=1,res=y_2\}$ y_1 $\{arg_1=y_2,arg_2=y_3,res=y_4\}$

Fig. 10 : Predicate Application



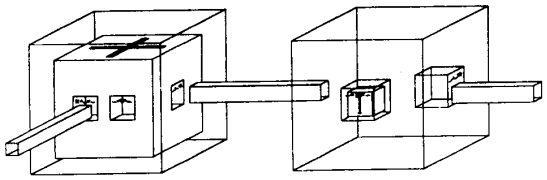
y_3 = *tree* $\{root=2,left=y_1,right=y_2\}$

Fig. 11 : Explicit unification

Atomic formulas or goals in CUBE can have two forms:

(a) A *predicate application* z $\{x_1=t_1,...,x_n=t_n\}$ $(n \geq 0)$. z can be either a reference cube referring to a predicate definition cube or a predicate-taking port in the current scope, or a holder cube taking a value of a predicate-type. The latter is represented by a transparent holder cube with ports $x_1,...,x_n$ on its sides (filled either with values or connected to pipes and filled with the inferred types), and a pipe supplying z connected to the holder cube. Fig. 10 shows the two forms. In both cases, the application must be well-typed with resulting type o. If the application fails, the cube fails.

(b) An explicit unification $z = t$. This is represented by a holder cube representing z filled with a value cube representing t (see Fig. 11). If t is of type τ, then z is of the same type. $z = t$ is of type o. If the unification fails, the cube fails. In Fig. 9 we have seen how this kind of atomic formula is used to describe curried application of constructors. The same technique can be used for predicates, as shown in Fig. 12.



y_2 = *plus* $\{arg_1=y_1\} \wedge y_2$ $\{arg_2=1,res=y_3\}$

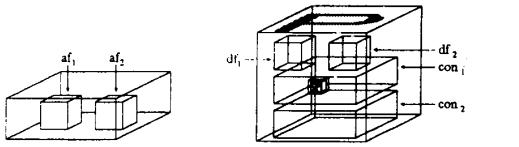
Fig. 12 : Curried predicate application

Pipes are used to connect holder cubes (or ports as special cases of holder cubes). Two connected holder cubes must be of compatible type. Upon execution, their values are unified (one can think of unification as of bidirectional dataflow). If they do not unify, and they are both on the same level, they both fail, if they are on different levels, only the inner one fails.

A *plane* is of the form

$con :: df_1 \wedge ... \wedge df_m \wedge af_1 \wedge ... \wedge af_n$

It is represented by a plane surrounding the local definition cubes $df_1,...,df_m$ and the n cubes representing the atomic formulas $af_1,...,af_n$. These cubes may be connected by pipes, and pipes may leave the plane as well. Each plane corresponds to a clause in a Prolog program. A plane fails if any of the cubes inside it fail. It succeeds if all of them succeed. Fig. 13.a shows a plane.



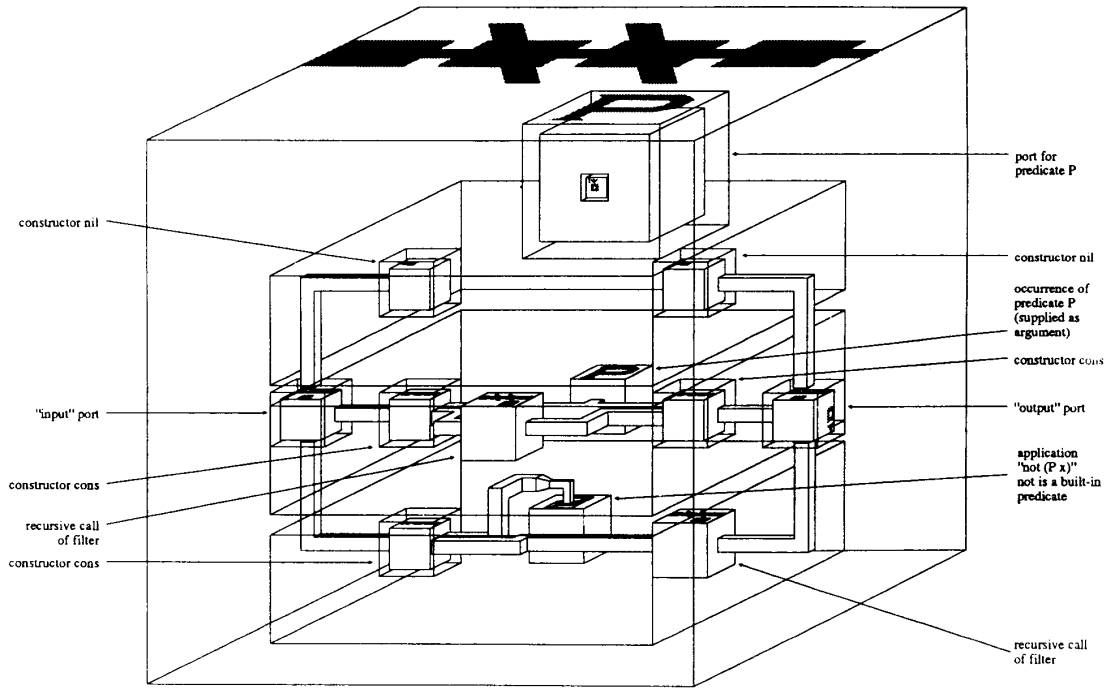
$(af_1 \wedge af_2)$ $p = \lambda\{x{:}T\}.df_1 \wedge df_2 \wedge (con_1 \vee con_2)$

Fig. 13 : Plane and Predicate Definition

Predicates are defined by *predicate definition cubes.* A predicate definition *pdf* is of the form

$p = \lambda\{x_1{:}\tau_1,...,x_k{:}\tau_k\}.df_1 \wedge ... \wedge df_m \wedge (con_1 \vee ... \vee con_n)$

It is represented by a predicate definition cube with icon p on its top, and ports named $x_1,...,x_k$ on its walls. The type inference system will determine the types of the ports, and fill



filter = $\lambda\{in$:*List* $\{X=\alpha_1\}$,*out*:*List* $\{X=\alpha_1\}$,*P*:$\{in:\alpha_1\} \rightarrow o\}$.$(in=nil \wedge out=nil)$
$\vee$ $(in=cons\ \{h=y_1,t=y_2\} \wedge P\ \{in=y_1\} \wedge filter\ \{in=y_2,out=y_3,P=P\} \wedge out=cons\ \{h=y_1,t=y_3\})$
$\vee$ $(in=cons\ \{h=y_4,t=y_5\} \wedge not\ \{in=P\ \{in=y_4\}\} \wedge filter\ \{in=y_5,out=out,P=P\})$

Fig 14 : *filter* — a higher-order predicate

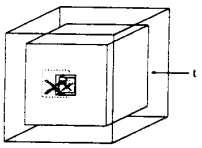
them with type cubes $\tau_1,...,\tau_k$. A predicate definition cube can contain other local (predicate or type) definition cubes $df_1,...,df_m$, which are arranged horizontally. It will also contain a number of disjoint "clauses" $con_1,...,con_n$, represented by planes. Fig. 13.b shows a predicate definition cube. Given above predicate definition, an application $p\ \{x_1=t_1,...,x_n=t_n\}$ succeeds if, upon unifying each x_i with t_i, any plane con_j succeeds. It fails if all con_j fail.

A CUBE *program* has the form $df_1 \wedge ... \wedge df_m \wedge af_1 \wedge ... \wedge af_n$. It is represented like a plane, except that there is no plane surrounding the local definitions and the atomic formulas. It succeeds if all af_i succeed, and it fails if any of them fails.

Note that, unlike to Prolog, in CUBE there is no ordering between atomic formulas or between planes. They are not evaluated in sequence, but rather concurrently.

It is a distinguishing feature of logic programming that a predicate application can succeed several times, each time obtaining different argument instantiations. For example, given a predicate $p = \lambda\{out:Int\}.out=1 \vee out=2$, the application $p\ \{out=x\}$ will succeed once with x bound to 1, and once with x bound to 2.

Fig. 14 shows an example of a higher-order predicate: the logic equivalent of the well-known *filter* functional, which takes a list l of τs and a predicate P of type $\{in:\tau\} \rightarrow o$, and returns a list of all those elements of l for which P holds.



t $\{x'\leftarrow x\}$

Fig. 15 : Port renaming

As said before, the name of the port of a predicate (or constructor) is part of its type. This raises one problem. Suppose we want to apply a higher-order predicate, say *filter*, to a predicate P whose type is not $\{in:\tau\} \rightarrow o$, but $\{in':\tau\} \rightarrow o$. We could of course define a new predicate $P' = \lambda\{in':\tau\}.P\ \{in'=in\}$. In order to avoid this extra definition, we introduce the concept of a *port renaming*. Given a term t of type $\{x_1:\tau_1,...,x_n:\tau_n\} \rightarrow \tau_0$, the type of $t\ \{x_1'\leftarrow x_1,...,x_n'\leftarrow x_n\}$ is $\{x_1':\tau_1,...,x_n':\tau_n\} \rightarrow \tau_0$. A port renaming is visualized by a transparent cube surrounding the cube representing t, and a transparent icon x_i' over each port name x_i. Fig. 15 shows a port renaming.

CUBE has a number of predefined predicates, eg. *greater*,...,*plus*,...,and *not*. Some of these predicates will, when applied, suspend until their arguments are sufficiently

known. For instance, *not* takes a nullary predicate p (a value of type o) as argument, but will resolve only when p is ground. An application of a non-constant predicate will resolve only when the predicate is ground. *plus* takes three arguments a,b,c of type *Num*, and computes $c = a+b$, but will resolve only when at least two of them are ground. Similar mechanisms have been used in Concurrent Prolog [17] and Parlog [7].

It is worth mentioning that the language we described so far is not only much more expressive than Prolog, but also considerably cleaner. It does not rely on clause- or goal-ordering, it is side effect-free, and does not contain extra-logical constructs like Prologs "cut". We believe that it is expressive enough to serve as a high-level general-purpose language.

Currently we are investigating how to include modules into CUBE, and how to incorporate a file system without sacrificing the purity (namely the confluence) of the language.

4 The Programming Environment

CUBE programs shall eventually be developed in a Virtual-Reality programming environment, which allows the user to directly manipulate the objects constituting the program, simply by grabbing, moving, and positioning them. Virtual Realities were first explored by researchers at NASA AMES. An overview of some of their research can be found in [6]. The hardware of a Virtual-Realitiy (VR) system consists of a set of position- and orientation-sensitive twin computer monitors mounted in front of the user's eyes that provide a stereoscopic view of computer-generated images, and a pair of gloves that can detect both their spatial position and orientation and the bending of the individual fingers. The software of a VR system generates a three-dimensional visual representation (called the Virtual Reality) of some objects, and allows the user to interact with this VR either by directly grabbing, moving, or otherwise manipulating the virtual objects, or by giving commands using a defined gesture protocol. Virtual Reality hardware is commercially available by now, and a number of applications have been developed for them [1][2].

5 Implementation

The implementation of CUBE is so far still in its infancy. At this point, we have implemented (in Prolog) a translation scheme which translates a subset of CUBE into a textual representation, and we have a working interpreter for an untyped version of the textual version of CUBE (written in Lazy ML). We are currently implementing a renderer to visualize CUBE programs, and plan next to integrate the different components.

6 Conclusion

We have informally described CUBE, a new language with a three-dimensional visual syntax, designed for a virtual-reality based programming environment. CUBE has a static polymorphic type system, and supports type inference. It is based on higher-order Horn Logic, thereby treating predicates as first-class values and allowing for higher-order predicates. It also allows the definition of local types and predicates, i.e. brings the concept of scope and nesting to logic programming. CUBE's semantics is inherently concurrent, allowing a parallel implementation.

Bibliography

[1] Chuck Blanchard et al., Reality Built for Two: A Virtual Reality Tool, in: *ACM Computer Graphics*, March 1990, pp. 35–36

[2] F. P. Brooks, Walkthrough – A Dynamic Graphics System for simulating Virtual Buildings, in: *Proc. 1986 Workshop on Interactive 3D Graphics*, pp. 9–12

[3] W. F. Clocksin, C. F. Mellish, *Programming in Prolog*, Springer-Verlag, Heidelberg, Germany, 1981

[4] V. S. Costa, D. H. D. Warren, R. Yang, Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism, in: *SIGPLAN Notices*, July 1991

[5] Luis Damas, R. Milner, Principal type-schemes for functional programs, in: *9th ACM Symp. on Principles of Programming Languages*, 1982, pp. 207–212

[6] James D. Foley, Interfaces for Advanced Computing, in: *Scientific American*, Oct. 1987, pp. 126–135

[7] Steve Gregory, *Parallel Logic Programming in Parlog – The Language and its Implementation*, Addison-Wesley, 1987

[8] G. P. Huet, The Undecidability of Unification in Third-Order Logic, in: *Information and Control*, Vol. 22, 1973, 257–267

[9] Takayuki D. Kimura, J. W. Choi, J. M. Mack, *A Visual Language for Keyboardless Programming*, Tech. Report WUCS-86-6, Dept. of Computer Science, Washington Univ., St. Louis, MO, March 1986

[10] Robert A. Kowalski, *Logic for Problem Solving*, North-Holland, 1979

[11] Dale A. Miller, Gopolan Nadathur, Higher-Order Logic Programming, in: *3rd Intl. Conf. on Logic Programming*, Springer LNCS #225, 1986, pp. 48-462

[12] Robin Milner, *A Proposal for Standard ML*, Report CSR-157-83, Computer Science Dept., Edinburgh Univ., 1983

[13] A. Mycroft, R. A. O'Keefe, A polymorphic type system for Prolog, in: *Artificial Intelligence*, No. 23, 1984, pp. 295–307

[14] Marc Najork, Eric Golin, Enhancing Show-and-Tell with a polymorphic type system and higher-order functions, in: *1990 IEEE Workshop on Visual Languages*, pp. 215–220

[15] Balkrishna Ramkumar, *The ROLOG User Manual – Version 1.0*, Dept. of Computer Science, Univ. of Illinois, Oct. 1989

[16] Uday S. Reddy, T. K. Lakshman, Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System, to appear in: *1991 International Logic Programming Symposium*, Oct. 1991

[17] E. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter*, Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Feb. 1983