A Prototype Implementation of the CUBE Language *

Marc A. Najork Simon M. Kaplan
Dept. of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

CUBE is a three-dimensional, visual, statically typed, inherently concurrent, higher-order logic programming language, aimed towards a virtual-reality-based programming environment. This paper describes a prototype implementation of CUBE.

1 Introduction

CUBE [7, 9] is a new programming language which combines several innovative features, namely

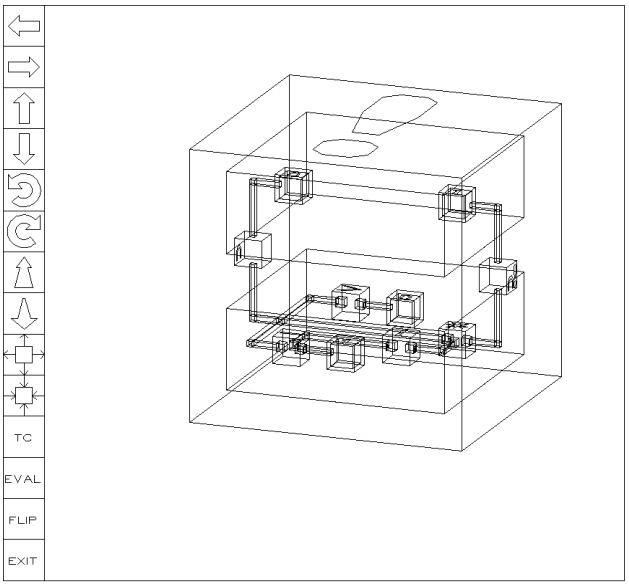
- a visual, three-dimensional syntax, which shall eventually make it possible to edit programs in a virtual-reality-based programming environment.
- a static, polymorphic type system, as used by many functional languages
- an inherently concurrent, higher-order Horn logic based semantics

CUBE was strongly influenced by Show and Tell [4], a visual functional language based on data flow, completion and consistency. CUBE's type system and higher-order aspects were inspired by ESTL [8]. Other visual logic languages include the Transparent Prolog Machine [2], Senay's and Lazzeri's system [10], pictorial Janus [3], and VLP [5]. Three-dimensional pictures have been used by Lieberman to visualize the execution of Lisp programs [6].

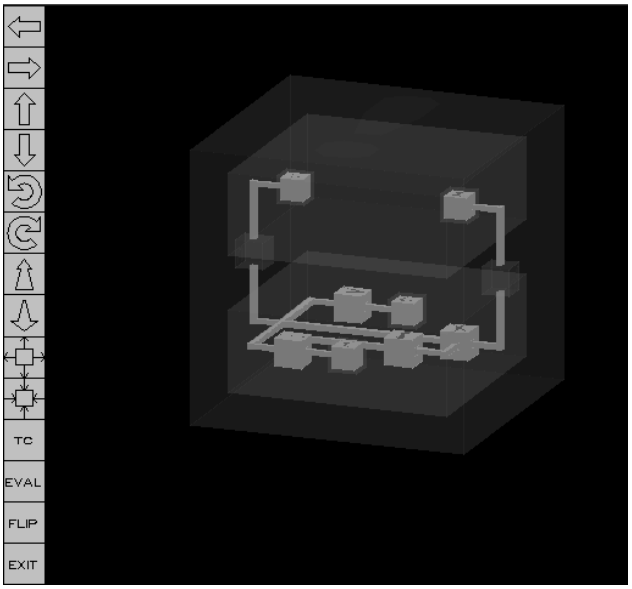
2 System Overview

The CUBE prototype system reads the description of a CUBE program from a file, renders it, and allows the user to "move around" in the program (see Fig.2). Upon request, it type-checks the program, renders the program together with the inferred types, and again allows the user to move around in it (see Fig.3). Upon further request, it evaluates the program, and allows the user to move around in the result (see Fig.4). The largest deficiency of the system is that it does not yet contain an editor; programs are supplied in form of hand-written text files.

The system consists of two programs: the Front-End, a C program responsible for rendering and user interaction,



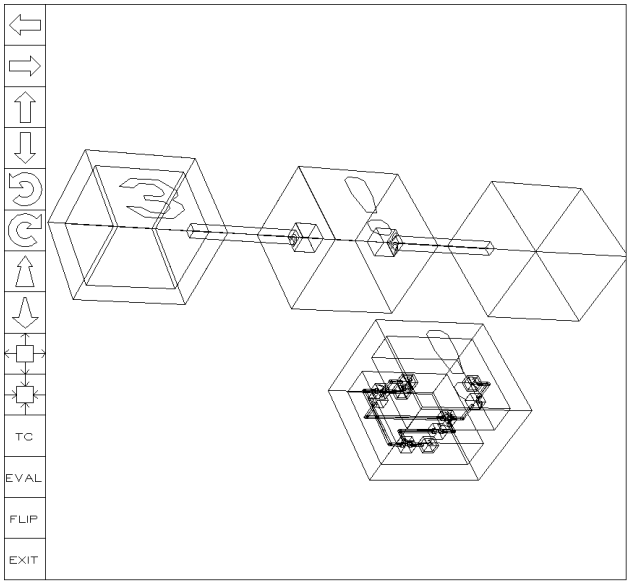
(a) Wireframe rendering



(b) High-quality rendering (original in color)

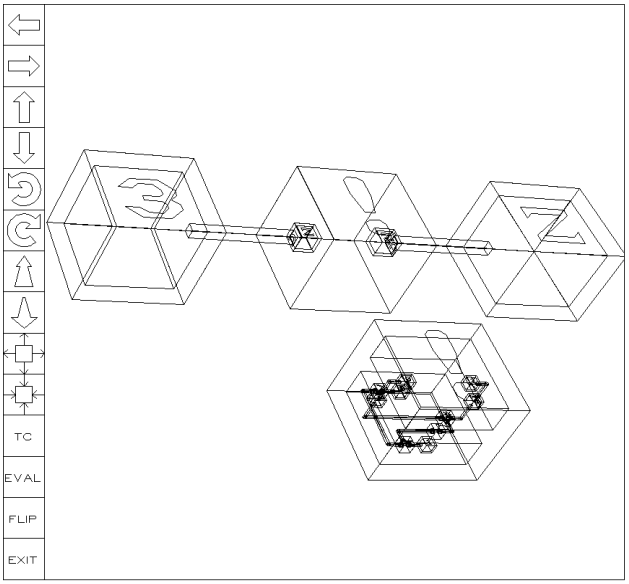
Figure 1: Renderings of the factorial predicate

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



Rendering of the program for computing the factorial of 3. Meaning of the buttons, from top to bottom: turn left/right, turn up/down, rotate counterclockwise/clockwise around view-axis, move forward/backward, zoom in/out, typecheck, evaluate, change rendering quality, and exit.

Figure 2: The factorial program before typechecking



The right holder cube contains the inferred result type, `Int`. Every initially empty holder cube is filled with its inferred type.

Figure 3: The factorial program after typechecking

and the Back-End, a Lazy ML program responsible for everything else. These two programs run concurrently, and communicate over Unix streams.

The system was implemented in such a way in order to combine fast rendering with rapid prototyping. The rendering step is presently the performance bottleneck, so it was mandatory to implement it in a fast, low-level language such as C. On the other hand, the rendering routines comprise just a small portion of the system, the less time-critical parts could still be implemented in a high-level language such as Lazy ML.

2.1 The Front-End

The Front-End of the CUBE system performs two tasks: it renders three-dimensional pictures transmitted from the Back-End onto an X window, and it detects mouse events, and either forwards them to the Back-End, or changes the viewpoint of the picture.

The picture description transmitted from the Back-End is on a fairly low level: polygons in three-space with attached color- and transparency-values.

The Front-End displays a CUBE program either as a wire-frame rendering, or it will use a more complex technique, which performs hidden-surface removal, and also handles transparent surfaces (see Fig.1). This technique combines concepts of z-buffering and α-channels, and delivers very realistic pictures, but unfortunately is very slow without hardware support.

2.2 The Back-End

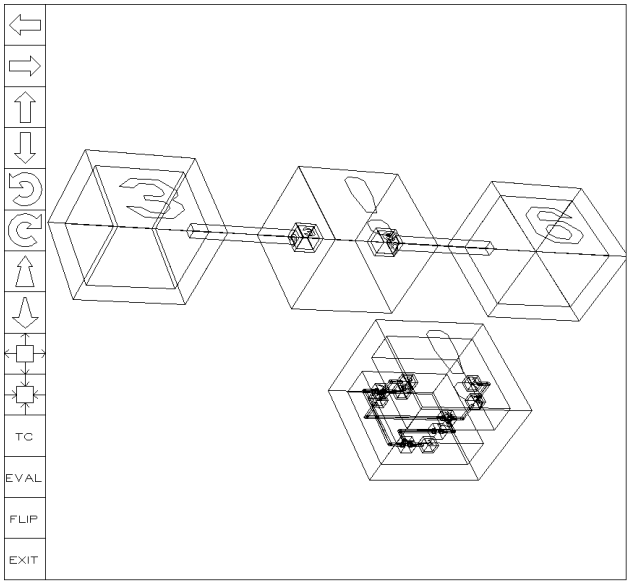
The Back-End of the system performs three major tasks: typechecking of programs, evaluation of programs, and visualization of programs, types, and results, i.e. translation of highly structured objects into polygons, which are then forwarded to the Front-End.

CUBE uses Hindley-Milner style type inference [1]. In order to infer the types of the subexpressions of a program, it is first translated into a simpler textual form, upon which fairly conventional type inference rules are applied. Details can be found in [7].

Semantically, CUBE is a higher-order Horn logic language. Predicates are viewed as special kinds of terms, and variables may range over predicates. However, we use normal first-order unification. Two predicates unify not if they describe the same relation (which is in general undecidable), but rather if they have unifying definitions.

CUBE is inherently concurrent. Our interpreter simulates a concurrent execution by maintaining a queue of processes (called a *configuration*), and by allowing each process a certain time-slice. In our terminology, a *process* consists of a store and a set of threads. A *thread* is a "lightweight process", which shares a store with other concurrent threads within the same process. Associated with each logic variable is a *location*. A *store* maps locations to values. A *value* can either be a term (the variable is "instantiated"), or be undefined (the variable is "uninstantiated"). Attached to each undefined value is a set of *wait-tokens*.

A thread corresponds to a goal to be proven, a set of threads within a process to a conjunction of subgoals, a



The right holder cube contains the result, namely 6.

Figure 4: The factorial program after evaluation

process to a particular proof attempt, and a configuration to different proof attempts.

One evaluation step consists of picking one thread out of a process, and resolving it. But not every thread (every goal) can be resolved right away. For instance, the goal "`plus 5 x y`" cannot be resolved, as long as both `x` and `y` are unknown. In such a case, the thread is suspended, and wait-tokens identifying it are attached to `x` and `y`. When a variable is instantiated with a value, all of the suspended threads with wait-tokens attached to the variable are resumed.

Before resolution, predicate applications have to be *reduced* to a normal form, which entails the binding of actual to formal parameters for applications.

CUBE is a higher-order language, which means that predicate and constructor applications may have variables in functor positions. If the functor is an uninstantiated variable, the reduction cannot be performed. The current thread is suspended, and a wait-token identifying it is attached to the variable.

A goal may be a unification of two expressions. Such a goal is resolved by first reducing both expressions to normal forms, and then unifying these.

A goal might also be reduced to the body of a predicate definition, i.e. a disjunction $e_1 \vee \cdots \vee e_m$, where each e_i is in turn a conjunction $e_{i1} \wedge \cdots \wedge e_{in_i}$. In this case, the current process P is replaced by m duplicates $P_1, \cdots, P_m$, and to each process P_i, new threads $e_{i1}, \cdots, e_{in_i}$ are added.

CUBE has a primitive negation predicate, which is cleaner than Prolog negation, as it suspends until the goal to be negated is completely ground. Evaluating a negation involves establishing a – again concurrent – subconfiguration, and determining if all its processes fail (i.e. the negation succeeds), or if at least one process succeeds (i.e. the negation fails).

3 Future Work

The prototype system demonstrates the feasibility of a CUBE implementation. In order to provide a useful system, three basic improvements have to be made:

- A visual editor for CUBE programs is needed. This editor should be structural and be context-sensitive, i.e. it should not allow the construction of ill-formed or ill-typed programs.
- In order to provide a truly interactive environment, the system's performance has to be increased. We hope to achieve this by porting our system onto a Silicon Graphics workstation.
- Our long-term goal is to port CUBE to a virtual reality environment to provide a better support to interact with its three-dimensional pictures than a mouse-based interface can provide.

References

[1] L. Damas, R. Milner. Principal type schemes for functional programs. In *9th ACM Symp. on Principles of Programming Languages*, pp. 207 – 212, 1982.

[2] M. Eisenstadt, M. Brayshaw. The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277 – 342, Dec. 1988.

[3] K. Kahn, V. Saraswat. Complete visualizations of concurrent programs and their executions. In *1990 IEEE Workshop on Visual Languages*, pp. 7–15.

[4] T. Kimura, J. Choi, J. Mack. A visual language for keyboardless programming. Tech. Rep. WUCS-86-6, Dept. of Comp. Science, Washington Univ., St. Louis, MO, March 1986.

[5] D. Ladret, M. Rueher. VLP: a Visual Programming Language. *Journal of Visual Languages and Computing*, 2(2):163 – 189, June 1991.

[6] H. Lieberman. A three-dimensional representation for program execution. In *1989 IEEE Workshop on Visual Languages*, pp. 111 – 116.

[7] M. Najork. Design and implementation of the Cube language. Unpublished manuscript, April 1992.

[8] M. Najork, E. Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *1990 IEEE Workshop on Vis. Lang.*, pp. 215 – 220.

[9] M. Najork, S. Kaplan. The Cube Language. In *1991 IEEE Workshop on Visual Languages*, pp. 218–224.

[10] H. Senay, S. Lazzeri. Graphical representation of logic programs and their behavior. In *1991 IEEE Workshop on Visual Languages*, pp. 25 – 31.