Geometric Programming:
A Programming Approach
to Geometric Design

ALBERTO PAOLUZZI
Terza Università di Roma

VALERIO PASCUCCI, and MICHELE VICENTINO
Università La Sapienza

This article presents a functional programming approach to geometric design with embedded polyhedral complexes. Its main goals are to show the expressive power of the language as well as its usefulness for geometric design. The language, named PIASM (the Programming IA LANGUAGE for Solid Modeling), introduces a very high level approach to "constructive" or "generative" modeling. Geometrical objects are generated by evaluating some suitable language expressions. Because generating expressions can be easily combined, the language also extends the standard variational geometry approach by supporting classes of geometric objects with varying topology and shape. The design language PLASM can be roughly considered as a geometry-oriented extension of a subset of the functional language FL. The language takes a dimension-independent approach to geometry representation and algorithms. In particular it implements an algebraic calculus over embedded polyhedra of any dimension. The generated objects are always geometrically consistent because the validity of geometry is guaranteed at a syntactical level. Such an approach allows one to use a representation scheme which is weaker than those usually adopted in solid modelers, thus encompassing a broader geometric domain, which contains solids, surfaces, and wire-frames, as well as higher-dimensional objects.

Categories and Subject Descriptors: D.1 [Software]: Programming Technique—applicative (functional) programming; D.3.2 [Programming Languages]: Language Classifications—applicative languages, design languages; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—curve, surface, solid and object representations; geometric algorithms, languages, and systems; hierarchy and geometric transformations; object hierarchy; J6 [Computer Applications]: Computer-Aided Engineering—computer-aided design, (CAD)

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Design language, functional modeling, geometric design, polyhedra, solid modeling, variational geometry

This work was partially supported from Italian Research Council, with contracts n. 92.02980.64 and n. 93.02528.64, within the "PF Edilizia" Project.


Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 1995 0730-0301/95/0700-0266 $03.50

Son of man, . . . , show them
the design and plan of the Temple, its exits and
entrances, its shape, how all of it is arranged, the
entire design and all its principles.
Give them all this in writing so that they can see
and take note of its design and the way it is all
arranged and carry it out.

EZEKIEL 43, 10–11

1. INTRODUCTION
This article discusses a functional programming approach to generative
geometric design. This approach aims to establish a sort of "calculus" over
geometric objects. A functional programming environment, where a direct
mapping is possible between expressions of the calculus and expressions of
the language, was a natural candidate for the implementation of such a
calculus. The presented language, based on dimension-independent represen-
tations with guaranteed geometric validity, leads to a versatile approach to
variational geometry, supporting classes of objects with varying topology and
shape. The programming approach to geometric computing proposed here
also retains the good properties of functional programming. In particular, it
allows one to write program code which is clear and concise.

The development of design projects is often approached with the aid of very
complex CAD systems based on design databases [Encarnação and
Schlechtendahl 1983; Encarnação and Lockemann 1990]. This approach is
largely diffused in the production environment, but it is both difficult and
expensive to maintain the geometric consistency of data, called validity
[Requicha 1980], when the project is subject to deep revision. The usual
solution is found by giving the designer some 2D/3D editing tools, which
interact with the design database in a consistent way. This approach may
allow a top-down development of the design but may be intricate and expen-
sive. The solution is particularly weak when the design must be frequently
and deeply reviewed. Conversely, the use of some database technology is
really necessary when managing large scale design projects, which may
involve hundreds of thousands of components. In particular, it is very useful
to store only a class representation of repeated parts, as well as to store the
contact relationship between any pair of connected components.

Last generation CAD systems, based on variational geometry [Gossard and
Lin 1983; Hoffman and Jaran 1992; Light and Gossard 1982; Yamaguchi and
Kimura 1990], approach the problem from a new perspective, and allow the
user to perform only a minimal set of corrections on a design prototype
[Hoffmann 1992]. Such systems assure the validity of geometric data either
by using some suitable machinery of constraint satisfaction [Steele 1980;
Leler 1988, 1985], or by solving large systems of nonlinear equations [Light
and Gossard 1982; Yamaguchi and Kimura 1990]. In particular, the user has
to specify the geometric properties (distances, tangencies, orthogonalities,
etc.) of design elements which must remain invariant when the shape is updated. The constraint-solving approach may be convenient when the design process is mainly bottom-up, as in the mechanical design, but may be slow and costly when the design must be developed in a top-down manner, as well as when some early design decisions are subject to change. When the object topology is modified, many constraints may be invalidated, in fact, and require a new specification. This problem is of great importance in building design, where almost all the updates induce some local or global changes to the object topology. When topology does not change, variational geometry offers the designer the best tool known today to describe part families and to automatically generate the appropriate shape instance for a specified context.

Quite recently, a novel powerful approach to shape specification, denoted as “generative modeling,” was proposed in [Snyder 1991, 1992; Snyder and Kajiya 1992]. Here shapes are described procedurally, in a fashion similar to other procedural methods in computer graphics (see e.g., the PostScript language [Adobe Systems Inc. 1985]). The power of this approach comes from the use of a number of operators able to combine shapes, mainly parametric curves, to generate a great number of different kinds of surfaces and deformable solid models. Like CSG, it supports a property of closure, so that other operators can be applied to shapes generated by some expression in the language, resulting in new and more complex shapes. This approach also supports parameterized and multidimensional shapes, as well as combinations of representational styles, including deformations, implicit descriptions, and set operations. Such a generative approach gives a natural, compact, and editable representation to a large class of curved objects. It is implemented as a special-purpose C language extension.

A functional programming approach to geometric design is described in this article. Language scripts are used to parametrically generate shape instances and to formalize the sequence of geometric design decisions, possibly with the aid of an interactive user interface. In this approach a complicated design may be described hierarchically and developed either top-down or bottom-up, or even by using some mixed strategy. Using a syntactically validated programming approach the design updating becomes easier in both cases: when the changes concern some specific components and when they apply to the overall design organization. In either case it is sufficient to update some program units at suitable hierarchical levels. It can be noted that some program units, which are either evaluated more frequently or are formalized more elegantly, may be abstracted as operators and may enter a design knowledge base (i.e., some specialized PLASM package). Later on, they can be reused easily by the same designer or by other persons on the project team during the development of different design projects.

In the authors’ view, in order to exploit the advantages of a geometric programming approach, the design language must satisfy the following requirements:

—Object abstraction. A complex design should be described in such a way that a unique definition of components instantiated in different positions...
exists, so that any correction in a class of components would automatically extend to every class instance within the design. Such an abstraction mechanism should be as powerful as possible, and should allow the definition of higher-level objects.

—Geometric validity. The geometric validity should always be guaranteed in every step of geometric design development and project review. Such a validity should be assured when small details of shape are updated and when the overall shape is subject to major changes. An object can be considered geometrically valid when a cell decomposition of the object exists, such that any point of the embedding space belongs to the interior of at most one cell and any cell is a face of some higher dimensional cell (neither self-intersection nor dangling parts are allowed). Notice that geometric validity does not imply the correctness of design choices, which remains the designer’s responsibility.

—Design encoding. The design decisions should be straightforwardly associated to language expressions. The design structure should be very close to the structure of the source script to be used as the “generating form” of the geometric shape. In particular, the user should be concerned only with the semantic meaning of the source script and not with the writing of efficient code. In other words, a script in a design language should express “what to do” rather than “how to do it.” Such an approach seems particularly important for architecture or engineering applications, where the designer is not expected to be a computer specialist as well.

Some simple mechanisms of data abstraction are currently supported by both CAD systems based on design databases [Encarnação and Lockemann 1990] and by standard graphics systems [Foley et al. 1990; Howard 1991]. The evaluation of higher-level functions instantiated with parametric objects has been introduced in PLASM. The requirement for geometric validity is satisfied here by making large use of implicit Boolean operations. The last requirement is satisfied by embedding the design language in a very high-level host language such as FL [Backus et al. 1990, 1989]. An optimizing compiler based on program transformation [Backus et al. 1989; Williams and Wimmers 1991] is available for this language.

PLASM takes a dimension-independent approach to geometry representation and algorithms. A programming approach to design and modeling means that many functions can be applied to geometric objects of varying dimensions. For instance, the function for axial composition of design parts given in Section 5 can be applied both to 2D and 3D shapes. Dimension independence also includes large conceptual savings. Because it needs internal data structures and geometric algorithms that are able to work in any dimension, it is induced to write smaller systems, which might be at the cost of some computational inefficiency. The advantages of representations and algorithms that can be applied to geometric objects of different dimensions are widely discussed in Paoluzzi et al. [1993], Rossignac and O’Connor [1990], Rossignac and Requicha [1991], and Snyder [1992]. The language is currently restricted to polyhedra and polyhedral approximations of curved objects. This is not
really a strong limitation, as every curved object described by parametric
equations can be generated as a piece-wise mapping of a cell-decomposed
polyhedral domain.

Checking for geometric validity is a difficult problem in geometric comput-
ing. Since Requicha's [1980] assessment of mathematical requirements for
data representations in solid modeling, it is well known that syntactic
checking of validity is rarely sufficient, and more expensive semantic checks
are often required (e.g., consider the equality test of two geometric objects). In
order to accommodate all the particular cases that may arise when perform-
ing solid operations, several intricate representations, which are quite
difficult to combine and to maintain, have been proposed for nonmanifold
boundaries of 3D solids. In a dimension-independent approach to representa-
tion of embedded objects almost every object (e.g., consider the 1D skeleton of
a 3D cue) is a nonmanifold, therefore the problem of maintaining a topologi-
cally consistent representation becomes even harder.

PLASM combines geometric objects without performing any consistency
checking, rather than adopting some generalized nonmanifold representation.
The latter choice would in fact imply: (a) the maintaining of a number of
cross-references between the subcomponents of any geometry and, even
worse (b) the constant possibility of topology invalidation by inconsistencies
due to numeric computations. The assumptions underlying our "weak" repre-
sentation of polyhedral complexes [Pascucci et al. 1994] are strong enough to
guarantee the validity of the representation, that is, the consistency of the
represented object. It is only required that an object is generated by some
syntactically well-formed expression in the language. The proper combination
of well-formed expressions will always give a well-formed result. In other
words, it is not possible to construct a geometrically invalid object. This is a
severe requirement that some geometric systems satisfy, and which is not
particularly original to PLASM (e.g., GENMOD uses interval analysis [Snyder
1992] for this purpose). We believe that our main ideas here were storing only
a minimal subset of adjacencies and adopting a policy of lazy evaluation of
geometry using cumulative Booleans.

This article is organized as follows. Section 2 motivates the choice of the
functional paradigm for geometric programming. In Section 3 some back-
ground is given about the FL approach to functional programming and about
the PLASM language. In Section 4 the geometric aspects of PLASM are
presented. In particular, brief descriptions of the representation scheme used
in the language and of the more useful geometric functions are given. Some
important aspects of geometric programming are also discussed in this
section, including the use of algebraic identities as rewriting rules and
constraint-based programming. A set of simple but nontrivial examples within
the domain of the architectural design is given in Section 5. The evaluation
mechanism of PLASM, which implements our approach to geometric comput-
ing, trading between storage of values, and evaluation of forms is outlined
briefly in Section 6. An alphabetic glossary of predefined PLASM functions is
displayed in the Appendix in order to make the examples discussed in this
paper easier to understand.
2. FUNCTIONAL PROGRAMMING AND GEOMETRIC DESIGN

The goal of this section is to motivate our choice of the functional paradigm for programming in geometric design. First we briefly recall some general features of functional programming, then we discuss some relationships between functional programming and geometric design.

Functional programming enjoys several good properties. The set of rules describing the syntax and the semantics of functional languages is very small, and each rule is very simple. This allows one to write code which is concise and clear. The meaning of a program is well understood, because no state is involved in program execution. Most of the advantages of functional programming come from the easiness of connecting programs in several different and sophisticated ways using concatenation and nesting. The diversity of functional programming mainly comes from using the functions both as programs and as data. This provides the powerful abstraction constituted by higher order functions, that is, functions that return other functions as results. Higher order functions are very useful in a design language, where it may be required to combine some basic geometric functions to provide a new set of geometric operators (see e.g., Section 4.4). Programming in a functional language is also naturally driven to modularity. Moreover, conversely to procedural languages, the parallel execution of a functional program does not require an explicit indication of what parts of code can be computed in parallel. Notice that parallelism may be very important in geometric computing, for example, in application areas such as building design, where very large scale models must be computed, which may require a tremendous amount of computing power.

A complex geometric shape, such as a building or a mechanical object, often results from the assembly of several parts which may be highly dependent on each other. In particular, each component may result from geometric computations involving some other components, and/or depending on the satisfaction of assigned constraints. In the authors' view, a functional approach seems a very appropriate way of modeling such an interaction. At each shape component a generating function is associated, which, in turn, depends on parameters which are other generating functions, the actual shapes of other components, or the numeric values of some intrinsic constraints. Circular constraints are not allowed in such an approach, and must be resolved by the designer. In this sense a programming approach to constraint-based design is less powerful than an approach based on constraint solvers. Conversely, the input to constraint solvers strongly depends on the actual shape. Actually, the two directions can be integrated into a single programming environment [Leler 1988].

Because geometric expressions may appear as actual parameters of functions, PLASM implements a programming approach to variational geometry. Geometric programming extends the standard variational geometry approach by supporting classes of objects with varying topology and shape. In fact the object resulting from the evaluation of a geometric expression will often depend on the values of other geometric expressions and functions. In this sense, a PLASM function can be seen as a shape prototype, or "generating
form" which is able to produce infinitely many different geometric objects, all with some common structure. This goes beyond the standard variational geometry approach, where all the instances from a family of constrained shapes have the same topology and where shape instances differ only on the numeric values of shape parameters. No type restriction is given to the parameters of PLASM forms: they can be functions, numbers, Booleans, polyhedra, mappings, or sequences of mixed types. Hence, the generative power of such an approach is very high.

It also must be said that inasmuch as designers are not required to be computer specialists, geometric computing may become dramatically inefficient, in particular when using functional programming. Nevertheless, an experienced programmer could significantly improve the code efficiency at the cost of producing a more cryptic text. Moreover, sophisticated compilation techniques can greatly speed up the execution of inefficient code. Systems which act on the source script by an automatic generation of improved programs are particularly useful in avoiding efficiency troubles. Examples of such compilation techniques are rewriting expression systems [Williams and Wimmers 1991] and program specialization methods [Jones et al. 1993].

3. BACKGROUND AND BASIC ELEMENTS
In this section, a brief outline of the FL approach to functional programming is given, together with an introduction to the design language PLASM. The interested reader is referred to the Backus' Turing lecture [Backus 1978] for motivation of programming at the function level, to Backus et al. [1990] and Williams [1982] for a more introductory treatment of the topic, and to Backus et al. [1989] for a complete description of FL syntax and semantics. A very interesting approach to 2D graphics with functional programming can be found in Lucas and Zilles [1988] and Zilles et al. [1988] where a functional system equivalent to PostScript is described. The PLASM language was introduced in Paoluzzi and Sansoni [1992].

3.1 FL Language and Primitives
FL (programming at Function Level) is an advanced language for functional programming developed by the Functional Programming Group of IBM Research Division at Almaden (California). It is a pure functional language based on combinatorial logic. It makes use of (both predefined and user-defined) combining forms, that is, higher-level functions that are applied to functions to produce new functions. The language introduces an algebra over programs—that is, a set of algebraic identities between functional expressions—for reasoning formally about programs. With this approach one may find simpler equivalent programs, both at design time and at compilation time, with great advantages in style and efficiency of program development [Williams and Wimmers 1991]. The FL approach allows one to easily combine programs already defined, so that new programs are obtained in a simple and elegant way. A discussion on program optimization, using program transformation in the context of a programming language, designed to facilitate the
writing of high level programs that are clear and concise, can be found in Williams and Wimmers [1991].

A small FL subset is described in the following. Primitive FL objects are characters, numbers, and truth values. Primitive objects, functions, applications, and sequences are expressions. Sequences are expressions separated by commas and contained within a pair of angle brackets. For example, \texttt{fun} is a function name, \texttt{1} and \texttt{-13.5} are numbers, and \texttt{<5, \texttt{fun}>} is a sequence. An application expression, \texttt{exp1:exp2}, applies the function resulting from the evaluation of \texttt{exp1} on the argument resulting from the evaluation of \texttt{exp2}. For example, \texttt{+:<1,3>} = \texttt{4}. Application associates to the left, that is, \texttt{f:g:h} = \texttt{(f:g):h}. Also, application binds stronger than composition (see the following): \texttt{f:g} = \texttt{h} = \texttt{(f:g) ~ h}. Binary functions can be used both in prefix and in infix form, so that it is possible to write \texttt{+.:<1,3>} as well as \texttt{1 + 3}. Some more important FL combining forms and functions are given in the following. In order to facilitate the reading of this paper, the PLASM syntax is used where it differs from the FL syntax.

\textit{Composition.} The binary form \texttt{~} is the standard mathematical operator for function composition:

\[ f \sim g \circ x = f \circ g \circ x \]

A COMP functional that composes a sequence of functions is also provided:

\[ \text{COMP}: \langle f_1, f_2, \ldots, f_n \rangle \circ x = (f_1 \sim f_2 \sim \cdots \sim f_n) \circ x = (f_1 \sim \cdots \sim f_n) : (f_n \circ x) \]

\textit{Construction.} The combining form CONS allows one to apply a sequence of functions to an argument producing the sequence of applications:

\[ \text{CONS}: \langle f_1, \ldots, f_n \rangle \circ x = \{ f_1, \ldots, f_n \} : x = (f_1 \circ x, \ldots, f_n \circ x) \]

For example, \texttt{cons:<+, -, >}, also written \texttt{[+, -]}, when applied to the argument \texttt{<3, 2>} gives the sequence of applications:

\[ [+,-]<3,2> = \langle +:<3,2>, -:<3,2> \rangle = \langle 5, 1 \rangle \]

\textit{Apply-to-all.} The combining form AA has the symmetric effect, that is, it applies a function to a sequence of arguments giving a sequence of applications

\[ \text{AA}: f: \langle x_1, \ldots, x_n \rangle = \langle f(x_1), \ldots, f(x_n) \rangle \]

\textit{Identity.} The function ID returns its argument unchanged, for any argument:

\[ \text{ID}: x = x \]

\textit{Constant.} The combining form K is evaluated as follows, for any \texttt{x2}:

\[ K:x1:x2 = (K:x1) : x2 = x1 \]

\textit{Conditional.} The conditional form, \texttt{IF:<p, f, g>}, where \texttt{p}, \texttt{f} and \texttt{g} are functions, is evaluated as follows:

\[ \text{IF}: \langle p, f, g \rangle : x = f : x \quad \text{if} \quad p : x = \text{TRUE} \]

\[ \text{IF}: \langle p, f, g \rangle : x = g : x \quad \text{if} \quad p : x = \text{FALSE} \]
Insert. The combining forms INSR (insert right) and INSL (insert left) allow one to apply a binary function to a sequence of arguments of any length:

\[
\begin{align*}
\text{INSR}: f: (x_1, x_2, \ldots, x_n) &= f: (x_1, \text{INSR}: (x_2, \ldots, x_n)) \\
\text{INSL}: f: (x_1, \ldots, x_{n-1}, x_n) &= f: (\text{INSL}: (x_1, \ldots, x_{n-1}), x_n)
\end{align*}
\]

where \(\text{INSR}: f: (x) = \text{INSL}: f: (x) = x\)

Catenate. The CAT function appends any number of input sequences creating a single output sequence:

\[
\text{CAT}: ([a, b, c], [d, e], \ldots, [x, y, w, z]) = [a, b, c, d, e, \ldots, x, y, w, z]
\]

Distribute. The functions DISTR (distribute right) and DISTL (distribute left) are defined as:

\[
\begin{align*}
\text{DISTR}: ([a, b, c], x) &= ([a, x], [b, x], [c, x]) \\
\text{DISTL}: (x, [a, b, c]) &= ([x, a], [x, b], [x, c])
\end{align*}
\]

Transpose. The function TRANS allows one to transpose a sequence of sequences. The name "transpose" is used because the function works exactly as a matrix transposition:

\[
\text{TRANS}: ([x_1, x_2, x_3], [y_1, y_2, y_3], [w_1, w_2, w_3], [z_1, z_2, z_3]) = \\
([x_1, y_1, w_1, z_1], [x_2, y_2, w_2, z_2], [x_3, y_3, w_3, z_3])
\]

3.2 The PLASM Language

PLASM, the Programming LAnguage for Solid Modeling, is a functional “design language,” being developed by the CAD Group at the University of Rome “La Sapienza” (Italy). PLASM, strongly influenced by FL, can evaluate polyhedral expressions, that is, expressions whose value is a polyhedral complex. PLASM can combine functions to produce higher-level functions in the FL style, so that it can be roughly considered as a geometry-oriented extension of a FL subset. Conversely, then, in FL, no free nesting of scopes and environments is allowed in a function definition, and no pattern matching is provided. A main difference with FL is it allows the association of identifiers to any language object and not only to functions.

3.2.1 Some Nongeometric Operators. The syntax of PLASM is very similar to that of a subset of FL. Some differences concern the meaning of a few characters and the optional use in a function definition of a specification list which precedes the list of formal parameters (see Section 3.2.2). Identifiers and keywords are given using a case unsensitive alphabet. The language allows for overloading of operators. For instance, + and * are used for addition and multiplication of numbers (e.g., \(3 + 2\)), for addition and multiplication of numeric functions (e.g., \((\sin + \cos):x\)), and for union and product of polyhedra (e.g., \(\text{pol1} + \text{pol2}\)), respectively. Some primitive nongeometric PLASM functions often used in this article are discussed in the sequel.

The function \# applied to a nonnegative integer \(n\) gives a repetition operator that can be applied to any PLASM expression. The result of the
double application is a sequence where the input expression is repeated \( n \) times:

\[
#:3: \text{expr} = \langle \text{expr}, \text{expr}, \text{expr} \rangle
\]

The function \( \#\# \) allows one to append (catenate) \( n \) instances of an input sequence. It is a shortcut for \( \text{COMP} \sim \langle \text{K:CAT}, \# \rangle \). Hence we have:

\[
\#\#:3: \langle a, b, c \rangle = \langle \text{COMP} \sim \langle \text{K:CAT}, \# \rangle :3: \langle a, b, c \rangle \rangle
\]

\[
= \text{COMP}:(\langle \text{K:CAT}, \# \langle a, b, c \rangle \rangle :3: \langle a, b, c \rangle)
\]

\[
= \langle \text{CAT} \sim (\#\#:3):\langle a, b, c \rangle, \langle a, b, c \rangle, \langle a, b, c \rangle \rangle
\]

\[
= \langle a, b, c, a, b, c, a, b, c \rangle
\]

A useful binary function \( \text{FromTo} \), also denoted with an infix \( ".." \), which generates integer sequences, works as:

\[
\text{FromTo:}\langle 1, 5 \rangle \text{ or } 1..5 = \langle 1, 2, 3, 4, 5 \rangle
\]

\( \text{LEN} \) is a primitive function which returns the integer \textit{length} of any input sequence. For instance:

\[
\text{LEN:}\langle a, b, c, d \rangle = 4
\]

\( \text{SEL} \) is a primitive \textit{generic selector} operator with a specification parameter \( i \), such that when applied to a sequence the \( i \)th sequence element is returned:

\[
\text{SEL:2:}\langle 13, 4.5, \text{id} \rangle = 4.5
\]

The functions \( \text{s}n \), where \( s \) is followed by any positive integer, denote as in FL the \textit{specific selector} operators, which return the \( n \)th element of an input sequence. Clearly, the following equivalence holds for any integer parameter:

\[
\text{s8 = SEL:8}
\]

Two combining forms \( \text{AC} \) (\textit{apply-in-composition}) and \( \text{AS} \) (\textit{apply-in-sequence}) are introduced to use a function with a number of specification parameters greater than the number used in the definition. For instance, by using the combining form \( \text{AC} \) and \( \text{AS} \), the \( \text{SEL} \) operator can work in two ways in order to get the behaviors desired in Paoluzzi and Sansoni [1992].

\[
\text{AC:SEL:}\langle 3, 1 \rangle: \langle \langle 1, 3, 8, 7 \rangle, 89, \text{fun} \rangle = 8
\]

\[
\text{AS:SEL:}\langle 3, 1 \rangle: \langle \langle 1, 3, 8, 7 \rangle, 89, \text{fun} \rangle = \langle \text{fun}, \langle 1, 3, 8, 7 \rangle \rangle
\]

Using \( \text{AC} \) and \( \text{AS} \), in the current PLASM syntax both kinds of specifications given in Paoluzzi and Sansoni [1992] are unified with the use of single curly bracket delimiters.

3.2.2 \textit{Function Definition}. The language interpreter interacts with a set of functions, here called the \textit{user functional environment}. When a definition is interpreted, a new operator is introduced in such an environment. A defini-
tion may contain a list of formal parameters. When a function without parameters is evaluated the first time, the PLASM interpreter produces and stores its value. In subsequent evaluations the stored value is directly returned.

In the current language syntax two kinds of functions are recognized: global (or top-level) functions and local functions. The global functions are recognized in a script by the presence of the reserved word DEF at the left side of their identifier. Global functions may contain a definition of local functions, to be given between the reserved words WHERE and END. The visibility of local functions is restricted to the scope of the global function where they are declared. A definition will contain the following:

1. The keyword "DEF";
2. the function identifier. User-defined function identifiers are specified using the case unsensitive alphabet \{A..Z,a..z,0..9, –, 1,A , ?\}, where, as usual, the first character should be nonnumeric;
3. the optional list of specification parameters, enclosed between \{" and "}\) brackets, to produce partially specified functions, giving the user further control over the behavior of the function;
4. the optional list of formal parameters, enclosed between \(" and \)\) brackets, with mandatory type specification. The type specification is constituted by a predicate separated from the parameter identifier(s) by \("::"\) terminal symbol;
5. the symbol " = ";
6. a PLASM expression which is the body of the function and allows one to compute its value. The body of a function will be called the generating form;
7. an optional set of local functions to be enclosed between the reserved words "WHERE" and "END." Local functions are used for increasing readability and for efficiency reasons;
8. an optional set of default value declarations for the specification and the formal parameters, to be enclosed between the reserved words "DEFAULT" and "END".

The syntax of PLASM definitions follows. Note that: (i) terminal symbols are written between double quotes using teletype characters; (ii) the notation \Object;\ denotes one or more \Object\ instances separated by \x\ symbols; (iii) square brackets denote optional parts.

```
Definition ::= "DEF Identifier[ Specifiers] [("Parameters;"\])
                   " = "Expression
                   ["WHERE" LocalDef; "END"]
                   ["DEFAULT" LocalDef; "END"]
Specifiers ::= ["Identifier ;"]
Parameters ::= Identifier ; "::" TypeChecker
LocalDef ::= Identifier " = " Expression
```

where \textit{TypeChecker} is any expression denoting a predicate, for instance, \text{IsIntPos}, \text{IsFun}, or \text{IsPolDim}:(2, 3).

The DEFAULT clause is used mainly for the sake of documentation. In fact, if all the specifiers and parameters are defined within the clause, then it becomes possible to ask the interpreter the default value of the function—useful as graphical or textual echo—for the purpose of quick documentation. If a default value was given for a formal specifier or parameter, then it is also possible to use this default by passing the keyword DF as the actual value of the parameter or specifier.

When a function definition is given with both \textit{Specifiers} and \textit{Parameters}, the function must be applied twice in order to return a value. The first application binds the formal specifiers to the actual ones. The second application binds the formal parameters to the actual ones. For instance, if the definition is

\begin{verbatim}
DEF Op \{i\} \{j::\text{IsIntPos}\} = +:(i, j),
\end{verbatim}

then \text{Op:1} returns a unary function, in this case the function \text{succ}: \text{Nat} \rightarrow \text{Nat}, whereas \text{Op:1:7} returns the number 8. Some remarks follow, with respect to the use of both specifiers and parameters in the PLASM syntax for function definition.

—A partially specified version, that is, a \textit{curried} version, of a function can be directly defined by using specifiers and parameters.

—A function can be defined by using only either specifiers or parameters. Specifiers are chosen if no type checking is requested, whereas parameters are used if type checking is needed.

—Even if a function definition is given using parameters only, then it is always possible (as in FL) to produce a curried function version.

The design decision of distinguishing between specifiers and parameters in PLASM was taken in order to promote the use of partially specified functions by the non-programmer user. As a matter of fact, the combination of curried functions results in more efficient programs, because objects of equal value (functions in this case) are shared, that is, computed only once. Finally, notice that if a function \textit{f} has only one argument, as in \text{DEF } f (a::\text{type1}) = \text{body}, then it must be applied without angle delimiters as \textit{f}\x. Otherwise, if the definition is \text{DEF } f (a1, \ldots , an::\text{type2}) = \text{body}, then the function must be applied to the sequence of actual arguments, as \textit{f}(x1, \ldots , xn). The same syntactical rule applies to the passing of actual specifier values.

4. PROGRAMMING WITH POLYHEDRA

PLASM is characterized by the use of dimension-independent complexes of polyhedra as primitive geometric objects. A polyhedral complex will be denoted here as an object of type “polyhedron.”
4.1 Representation of Geometry

A polyhedron is characterized by an intrinsic dimension $d$ and by an embedding dimension $n$. The first one equates the maximal number (decremented by one) of affinely independent vertices whose convex combination spans some polyhedron portion. The second one equates the number of coordinates of vertices. We call the dimension of an embedded polyhedron the ordered pair $(d, n)$ of its intrinsic and embedding dimensions. For instance, a plane polygon has dimension $(2, 2)$ and a space polygon has dimension $(2, 3)$. An object is considered "solid" when $d = n$.

The representation scheme [Requicha 1980] used within the language is defined on the domain of multilevel hierarchical collections of quasidisjoint polyhedra, called polyhedral complexes. The range of the representation scheme is the set of direct acyclic multigraphs. In a multigraph representation of a polyhedral complex the following kinds of elements should be noted:

(i) **Leaf node** (outdegree = 0). It is associated with an elementary polyhedron, represented as a triple $(F, V, C)$, where $F$ is a face set, $V$ is a vertex set and $C$ is a cell set. Cells in $C$ are linear polytopes (convex cells), bounded by face hyperplanes stored in $F$. Either a *face-based* or a *vertex-based* representation of a polyhedron can be used. In the first case, face hyperplane equations are explicitly stored, whereas vertices are only implicitly described as lists of pointers to incident hyperplanes. Conversely, in the second case, vertex coordinates are explicitly stored, whereas faces are described as lists of vertex pointers.

(ii) **Nonleaf node** (outdegree > 0). It is associated with a polyhedral complex. This complex is represented by the ordered multiset of (leaf and nonleaf) nodes pointed by the arcs outgoing from the node. The corresponding elementary polyhedra and polyhedral complexes should be quasidisjoint, that is, should be allowed to intersect only on the boundaries.

(iii) **Oriented arc.** It is associated with an affine transformation. According to standard graphic techniques, a proper product of the geometric information contained in a subgraph times the matrix associated to the arc entering the root node of the subgraph, allows one to transform the pointed object (complex or polyhedron) from local coordinates to the coordinates of the pointing complex.

Such a representation requires a double decomposition of the object geometry: (a) a decomposition of a complex in subcomplexes and polyhedra, and (b) a decomposition of a polyhedron in convex cells. The first level of decomposition is required by the user in order to structure the defined object with meaningful elements and subelements, as well as to handle them with local coordinate frames. The second level of decomposition is due to the capability of the system to process as basic elements only linear solid polytopes, that is, unredundant sets of linear inequalities. This system design decision allowed the implementation of most basic operators in PLASM using efficient and robust dimension-independent algorithms from computational geometry.
The only geometrical operator in PLASM that requires the user to consider this double decomposition is the MKPOL operator (see Section 4.2.2), which is a very low-level geometrical function used as constructor of polyhedral objects. The decomposition of elementary polyhedra in convex cells should not be considered by the user, which is usually required to adopt a higher level generative viewpoint.

Elementary polyhedra are always considered solid and stored as sets of inequalities. This means that the intrinsic dimension of polyhedra is always equal to the dimension of the linear space where they are embedded. For example, the 2-dimensional boundary of a 3D cube is represented by the multigraph shown in Figure 1(a), where the only elementary polyhedron is the 2D solid square. A local editing with solid methods (e.g., Boolean operations) of one of the six instances of the boundary face would result in the object of Figure 1(b).

Because all topological and geometrical information is stored in elementary polyhedra, a $V^*$ traversal algorithm, named forward evaluation, must be applied to such a multigraph representation in order to explicitly generate the associated polyhedral complex. The standard depth-first search traversal with a current transformation matrix [Foley et al. 1990] is used for this purpose, extended with a linear mapping (from $\mathbb{R}^d$ to a $\mathbb{R}^n$ coordinate subspace) of the elementary $d$-dimensional polyhedra to be embedded in $\mathbb{R}^n$.

We notice that such an approach allows one to handle open, closed, manifold, and nonmanifold polyhedra with extreme simplicity.

A more sophisticated and time consuming $V$ traversal algorithm, named backward evaluation, is required if the so-called weak representation is used. Such a weak representation is a directed acyclic multigraph that corresponds to a polyhedral sequence where the constraint of quasidisjointness has been released. In this case, in order to generate the associated geometrically valid (i.e., quasidisjoint) polyhedral complex, the traversal algorithm should breadth-first search the multigraph and selectively remove the intersecting portions of overlapping polyhedra. In particular:

(a) if $q$ is a leaf node, then $Q = q \cdot M$ is an (elementary) embedded polyhedron, where $M$ is a linear mapping performing a proper embedding of the “solid” object $q$;

(b) if $p$ is a nonleaf node and $(p_1, p_2, \ldots, p_k)$ is the ordered multiset of nodes pointed by $p$, then the (hierarchical) polyhedral complex $(P_1, P_2, \ldots, P_k)$ is computed, where:

(b1) the polyhedral complex $P_1 = \ast V(p_1) \cdot T_1$ is associated to the first node, with $T_1$ the affine transformation matrix associated to the arc from $p$ to $p_1$;

(b2) the polyhedral complex associated to the node $p_i$, $2 \leq i \leq k$, is then evaluated as $P_i = (\ast V(p_i) \cdot T_i) - P_{i-1} \cdots - P_1$.

Actually, in our current implementation, an embedding mapping can be associated not only to the arcs pointing to leaf nodes, but to every arc in the
multigraph representation. This allows one also to represent polyhedral sequences with objects of varying dimensions. Clearly, the implementation of progressive difference into the *V validating traversal algorithm becomes much more difficult and has not been accomplished. A representation of objects of mixed varying dimensions and an approach to nonregularized Boolean operations are described in Rossignac and O'Connor [1990] and in Rossignac and Requicha [1991], respectively.

The *V traversal is necessary for the STRUCT function described in Section 4.2.5. In particular, an example is shown in Figure 4, and a related discussion is given in Section 4.2.5 of such an evaluation algorithm of a polyhedral complex, starting from a weak representation associated with some suitable polyhedral sequence. We also remember that two different representations of

geometry that we call *face-based* and *vertex-based* may be used. A representation conversion is executed when necessary, depending on the function to be applied to the polyhedral complex. The *base-based* representation is the one more frequently used.

The advantages of the face-based representation with respect to robustness of geometric computations and to the ease of implementing some useful operations, convinced us to use the face-based as the main representation in the language. In such a representation the vertices are implicitly stored as unevaluated sets of incident-face hyperplanes. Hence some algorithms, for example, the MAX and the MIN operations (see Section 4.2.9), are implemented by using linear programming methods on such a representation. Even if the computation of the maximum/minimum coordinate of a set of low-dimensional polytopes is not actually a difficult task, such a computation is not trivial, if compared to the use of a vertex-based representation of cells. Anyway, if the vertex-based representation is the current one while computing such a function, then it is directly used.

### 4.2 Some Geometric Operators

Some predefined geometric PLASM operators are described in this section. Some of them were introduced in Paoluzzi and Sansoni [1992], where a wider discussion may be found. A generalized product of cell complexes which contains as special cases the standard intersection of cell-decomposed polyhedra, the finite and nonfinite extrusion, the intersection of extrusions, and the Cartesian product is discussed in Bernardino et al. [1993].

#### 4.2.1 Predicate Testing Dimensions of Polyhedra

*lsPolDim*(d,n) is an expression denoting a predicate that returns TRUE when applied to a polyhedron if its dimension is (d, n). The predicate returns FALSE if the argument is either not a polyhedron or has a different dimension. Note that in order to test if an expression gives a polyhedron of any dimension, the test predicate *IsPoI* can be directly applied to the polyhedron parameter. For instance, *lsPol:expr* has value TRUE if expr denotes a polyhedron. Integer predefined functions on polyhedra are *DIM* and *RN*. They give the intrinsic and the embedding dimensions, respectively, and return an error if the argument is not a polyhedron.

#### 4.2.2 Constructor Functions *MKPOL* and *UKPOL*

The predefined function *MKPOL* (MaKePOL) allows one to introduce user-defined polyhedral complexes as elementary objects. The input to such a function is a long three sequence of vertices, convex cells, and polyhedral cells. Vertices are given as sequences of coordinates; convex cells are given as sequences of vertex indices. Each cell is implicitly defined as the convex hull of its vertices. Polyhedral cells are given as sequences of convex cell indices, and are defined as the union of such cells. The geometric validity of the generated polyhedral complex as a quasidisjoint partition with polyhedral cells is guaranteed for any input. The validity is ensured by applying a *V* traversal to the sequence...
of polyhedral cells. Clearly, the validated polyhedral complex may differ from
the previous \( V \) validation polyhedral sequence.

A polyhedron of dimension \((0, n)\), that is, a set of points, results from the
application of the function \( MKPOL \) on triples where vertices are long \( n \)
sequences of real numbers, and where any convex cell is made of just one
vertex, and any polyhedral cell is made of just one convex cell. For instance,
the 0-skeleton (see Section 4.2.7) of the standard 3D simplex (i.e., the unit
tetrahedron) can be either generated by the expression \((\odot 0 \sim SIMPLEX):3\) or
directly defined as:

\[
\text{DEF tetraVertices} = MKPOL:\langle \text{vertices, convexCells, polyhedralCells} \rangle
\]
\[
\text{WHERE}
\]
\[
\text{vertices} = \langle (0,0,0), (1,0,0), (0,1,0), (0,0,1) \rangle
\]
\[
\text{convexCells} = AA:ID:[(1..4)]
\]
\[
\text{polyhedralCells} = AA:ID:[(1..4)]
\]
\[
\text{END}
\]

Such a \((0,3)\) dimensional polyhedron with 4 vertices is internally repre-
sented as a directed multigraph with two nodes and 4 arcs. The ending node
of all arcs is the 0-dimensional zero \( o^{0.0} \), represented in homogeneous coordi-
nates as \([1]\); each arc is associated with a \( 4 \times 4 \) translation matrix which is
applied to the origin of \( S^3 \) (\( o^{0.0} \) embedded in \( S^3 \)) using homogeneous
coordinates, that is, applied to the vector \((x_0, x_1, x_2, x_3) = [1 0 0 0]\).

Example 4.2.1. A simple example of the \( MKPOL \) constructor is given here.
In particular, a \((1,2)\) dimensional polyhedron, which models the schematic
layout shown in Figure 3(a), is defined. Notice that such an object can be built
easily by using an interactive input device.

\[
\text{DEF LayoutScheme} = MKPOL:\langle \text{vertices, convexCells, polyhedralCells} \rangle
\]
\[
\text{WHERE}
\]
\[
\text{vertices} = \langle (3,15), (6,15), (8,15), (13,15), (15,15), (17,15), (3,11),
\]
\[
(11,11), (13,11), (17,11), (5,8), (11,8), (3,3), (6,3), (8,3), (13,3),
\]
\[
(15,3), (17,3), (3,8), (3,10), (11,9), (11,13), (11,3), (11,15),
\]
\[
\text{convexCells} = \langle (1,2), (3,4), (5,6), (7,8), (9,10), (11,12), (13,14),
\]
\[
(15,16), (17,18), (13,19), (20,1), (23,21), (22,24), (6,10), (10,18) \rangle,
\]
\[
\text{polyhedralCells} = \langle (10,11,14,15), (1,2,3,7,8,9), (4,5,6,12,13) \rangle
\]
\[
\text{END}
\]

Three (unconnected) polyhedral cells are given, corresponding to the horizon-
tal and vertical envelope and to the internal partitions, respectively. Then a
product times a 1D polyhedron, built from PLASM primitives described in the
next subsections, is defined as \( \text{LayoutScheme} \ast \text{QUOTE}:(10) \) and shown in
Figure 3(b).

The reverse function with respect to \( MKPOL \) is called \( UKPOL \) (UnmaKe-
POL). Such a function, when applied to any polyhedral expression, returns a
triple of sequences corresponding to vertices, convex cells, and polyhedra
extracted from the multigraph representation of the evaluated polyhedral

expression. Even if MKPOL $\sim$ UKPOL is not equivalent to ID, the following algebraic identity holds:

$$\text{MKPOL} = \text{MKPOL} - \text{UKPOL} - \text{MKPOL}$$

4.2.3 JOIN and EMBED Functions. An important predefined geometric function on polyhedra is JOIN. It is evaluated either on a single polyhedron, even nonconvex, or on sequences of polyhedra. It returns a single polyhedron, computed as the convex hull of the vertices of the arguments. If the argument polyhedra are in different spaces, they are embedded in coordinate subspaces of the space of maximum dimension, where the result is also given. It is implemented in the prototype interpreter by means of the “Beneath-Beyond” Kallay’s algorithm [Kallay 1981] using the version discussed in Edelsbrunner [1987]. The same algorithm is used for the implementation of the MKPOL operation. The primitive function EMBED:m when applied to a polyhedron of dimension $(d, n)$ embeds it in the coordinate subspace $x_{n+1} = \cdots = x_{n+m} = 0$ of the space $\mathbb{R}^{n+m}$.

4.2.4 Affine Transformations. Predefined functions performing elementary affine transformations are denoted as T, S, R, H, which stand for translation, scaling, rotation, and shearing, respectively. Translation and scaling functions are applied to sequences of specifiers and parameters with equal length. The first sequence of integer specifiers denotes the set of coordinates modified by the transformation; the second sequence contains the real parameters of the transformation. For example, a translation along the first and third coordinate directions of 3.5 and 8/5 units, respectively, is given as:

$$T:(1,3):(3.5,8/5)$$

A rotation is always applied on double sequences of length two and one, respectively. As a matter of fact, we consider elementary rotation (in spaces of any dimension [Paoluzzi et al. 1993]) an isometry with a unit determinant that changes only two coordinates according to the usual pattern of $\sin$ and
cos functions. For example, a rotation in 3D of angle \( \alpha \) around the second coordinate axis is given as:

\[
R:(1,3):\alpha
\]

For efficiency reasons, a special form for parameter specification has been implemented for some frequently used elementary geometric functions. In particular, the functions MIN, MAX, SIZE, and BOX, described in the following, and the affine transformation functions T, S, R, H can be specified both on a single value and on a sequence of values. For example, T denotes translation, and T:3:2.5 means translation of 2.5 units on the third coordinate. Hence, for a translation on more than one coordinate we can write, for example, T:(1,3):(2.5,6/1.5). Note that several equivalences hold for such functions. For instance:

\[
T:(1,3):(2.5,6/1.5) = T:1:2.5 ~ T:3:(6/1.5)
\]

Transformations can only be applied to polyhedrally-valued expressions, or used within structures (see Section 4.2.5). A sequence of rotation and translation applied to the P polyhedron can so be written in two equivalent ways:

\[
(T:2:3.7 ~ R:(1,2):\alpha):P = STRUCT:(T:2:3.7, R:(1,2):\alpha, P)
\]

The result of both expressions is a polyhedral complex, that is, a directed acyclic multigraph, in this case just a linear graph, where P is associated to the ending node, and where proper transformation matrices are associated to the two arcs.

A function MAT allows the applying of a matrix mapping to a pol expression. Such a function is defined as follows:

\[
MAT:\text{matrix}:pol = STRUCT:(MAT:\text{matrix}, pol)
\]

where MAT:matrix is a transformation function which is accepted as a component of a structure, and matrix is a sequence of sequences of equal length of real numbers, that is, is a matrix description by rows. Some operators are available to manipulate such matrix sequences. For instance INV, TRANS, and * execute the inversion, transposition, and binary product of matrices, respectively.

4.2.5 STRUCT Function. A very important PLASM function is the STRUCT function, which is applied to sequences of pre- or user-defined affine transformation functions and polyhedra and returns a new (composite) polyhedron. The semantics of the STRUCT function is very close to that of PHIGS structures [Howard et al. 1991]. Such a function is able to define a hierarchical polyhedral complex by using affine transformations of coordinates and component complexes given in local coordinates. From a syntactical point of view, a STRUCT function must be applied to a sequence containing only expressions that denote polyhedra and transformation functions. From a semantic point of view, the result is obtained by building and traversing a multigraph representation of the sequence, where the transformation functions result in linear operators associated to the arcs. Each transformation is
implicitly applied to all the objects which follow it in the sequence, exactly as in PHIGS.

In order to guarantee the geometric validity of the output of a STRUCT application for any well-formed value of the input sequence, a progressive difference operation [Paoluzzi and Sansoni 1992] should always apply to the polyhedral elements of the sequence after they are transformed in the same system of coordinates. For example, if \( \langle \text{pol}_1, \text{pol}_2, \text{pol}_3 \rangle \) is such a transformed sequence, the result of \( \text{STRUCT} : \langle \text{pol}_1, \text{pol}_2, \text{pol}_3 \rangle \) is the polyhedral complex having was polyhedral cells \( \text{pol}_1, \text{pol}_2 - \text{pol}_1, \text{and pol}_3 - \text{pol}_2 - \text{pol}_1 \). Roughly, the STRUCT operator transforms a set covering in a set partitioning, allowing one to obtain an automatic shape detailing, which is particularly useful in the context of architectural CAD. The example in Figure 4 shows that different shape detailing can be obtained by simply changing the order of elements within the input sequence to a STRUCT application.

In other words, an implementation of the STRUCT function oriented to always produce geometrically valid objects would require the application of the \( *V \) backward evaluation algorithm (see Section 4.1) to the multigraph representation of the input sequence. Actually, a forward evaluation \( V^+ \) is often sufficient. In fact, a policy of lazy evaluation of geometry can be adopted in many cases [Pascucci 1993], where the more expensive \( *V \) algorithm is delayed as much as possible.

4.2.6 1D Polyhedra Generation. The QUOTE operator maps a sequence of nonzero reals in a 1D polyhedral complex. In particular, positive reals \( \langle x_1, \ldots, x_n \rangle \) are mapped into adjacent line segments of length \( x_i \). A negative number \( x_i \) is mapped into an empty interval of length \( |x_i| \). The QUOTE identifier is derived from the Italian word used to denote the linear dimen-
A, Paoluzzi et al.

Fig. 5. Constructor of 1D polyhedra. The result of the evaluation of the expression QUOTE: (2, 5, 2, -5, 2, 8, 2, -5, 2, 5, 2) is displayed.

Fig. 6. Boundary and skeleton extraction operations. (a) 2D polyhedral complex A; (b) Boundary @:A; (c) 1-skeleton @1:A; (d) Complex (internal partitions) generated by the expression (@1 − @):A = (− ~ [@1, @]):A.

sions of technical drawings. The following equivalence holds between the QUOTE function and the STRUCT function:

QUOTE = STRUCT ~ CAT ~ AA:IF:(GT:O, [cuboid ~ [ID], T:1], [T:1 ~ ABS])

where GT:O:x is a predicate testing if x > 0. The function cuboid, defined in Example 4.2.7, allows one to generate line segments (as well n-dimensional intervals).

Example 4.2.6. The result of the application QUOTE: (2, 5, 2, -5, 2, 8, 2, -5, 2, 5, 2) is displayed in Figure 5. We may define a parameterized function using a λ style:

DEF pol1D (a, b::IsRealPos) = QUOTE: (2, a, 2, -5, 2, b, 2, -5, 2, a, 2)

and then evaluate the expression pol1D: (5, 8).

4.2.7 Booleans, Skeletons and Products. The language, besides the regularized Boolean operations of union, intersection, and difference, denoted respectively as A + B, A & B, and A − B, provides some additional dimension-independent geometric operators, such as the extractor of the complex of k-dimensional cells, named k-skeleton @k (k = 0, 1, 2,...), and the boundary operator @ on a polyhedral complex (see Figure 6). A dimension-independent product *, offset ++, and intersection of extrusions && are also provided (see Figures 7, 15, and 16, and Paoluzzi and Sansoni [1992]). The dimension-independent algorithm for binary product and intersection of extrusions of cell-decomposed polyhedra is discussed in Bernardini et al. [1993]. A method for reducing the product on polyhedral complexes (represented as directed multigraphs) to the product on cell-decomposed elementary polyhedra is discussed in Paoluzzi et al. [1994].

Example 4.2.7. A function cuboid to generate 1D segments, 2D rectangles, 3D parallelepipeds, and higher dimensional hypercuboids may be defined as:

\[
\text{DEF cuboid} = (\text{INSR}: \ast) \sim (\text{AA}:\text{QUOTE \sim [ID]})
\]

When the cuboid function is applied on a sequence of length \(d \geq 0\) of positive reals, then a \(d\)-dimensional hypercuboid is returned. In fact we have, where \(p_1, p_2, \ldots, p_d\) are 1D segments:

\[
cuboid: (x_1, x_2, \ldots, x_d) = (\text{INSR}: \ast \sim \text{AA}:\text{QUOTE \sim [ID]}):(x_1, x_2, \ldots, x_d)
\]

4.2.8 MAP Function. The primitive function MAP is applied to a sequence of \(n \geq m\) real functions, written using the selectors \(s_1, \ldots, s_m\). The resulting function is then applied to a \((d, m)\)-polyhedron. The result of this double application is the \((d, n)\)-polyhedron obtained by applying the real functions to the polyhedron vertices. For instance, the embedding of the \((2, 2)\)-polyhedron \(P_{01}\) in the subspace \(z = 0\) of \(\mathbb{R}^3\) can be also written: \(\text{MAP}: [s_1, s_2, K:0]:P_{01}\). A more interesting use of the MAP function is given in Example 4.2.8. In order to guarantee that MAP will work properly in any case, a simplicial decomposition of the input polyhedron is needed. Hence the body of the MAP function performs the following actions: (a) a vertex-based representation of the cells of the input polyhedron is generated; (b) a simplicial decomposition of each cell by using a winged representation [Paoluzzi et al. 1993] is computed; and (c) the mapping function is applied to the vertices of such decomposition.

Example 4.2.8. A PLASM function which generates polyhedral approximations of 3D cylinders with a variable number \(n\) of lateral faces (see Figure 8) is here discussed. First, a regular 1D cell decomposition in \(n\) segments of the interval \([0, 2\pi]\) is generated. Second, a parametric mapping is applied, in order to obtain a piecewise linear approximation of the circle boundary. Third, the JOIN function is applied to give the (solid) 2D basis, and finally a product of the basis times the 1D unit segment gives the unit cylinder.
The function \( \text{SplitDomain} \), defined as

\[
\text{DEF SplitDomain} \ (n::\text{IntPos}) = \#::n:(2 \cdot \pi / n)
\]

where \( \pi \) is the PLASM denotation for \( \pi \), when applied to a positive integer \( n \) produces the sequence of \( n \) numbers \((2 \cdot \pi / n, \ldots, 2 \cdot \pi / n)\) whose sum is \( 2\pi \). Hence, the function

\[
\text{DEF Intervals} = \text{QUOTE} ~ \text{SplitDomain}
\]

when applied to a positive integer \( n \) gives a polyhedron of dimension \((1,1)\) with \( n \) cells of equal size in the interval \([0,2\pi]\). The primitive function MAP is then used in order to generate the boundary curve of the unit circle. The JOIN of the piecewise approximation of the circle boundary gives the corresponding approximation of the unit circle:

\[
\text{DEF Basis} = \text{JOIN} \sim \text{MAP}:[\cos \sim s1, \sin \sim s1] \sim \text{Intervals}
\]

Finally, a product of \( \text{Basis} \cdot n \) times the 1D polyhedron \( \text{QUOTE}:(1) \) produces the desired result:

\[
\text{DEF Cylinder} \ (n::\text{GE}:3) = \text{Basis} \cdot n \cdot \text{QUOTE}:(1)
\]

DEFAULT \( n = 24 \) END

### 4.2.9 MIN, MAX, BOX, and SIZE Functions

A MIN\(i::\text{pol}\) expression returns the minimum value of the \( i \)th coordinate of the points of the \( \text{pol} \) object. Analogously, MAX\(i::\text{pol}\) returns the maximum value. A function MED can be consequently defined:

\[
\text{DEF MED}\{i\}(x::\text{IsPol}) = (1 / 2) \cdot (\text{MAX} + \text{MIN}):i:x
\]

Two predefined functions, which make use of MIN and MAX, allow one to compute the containment BOX and the SIZE of any polyhedrally-valued expression expr. In the first case the minimal cuboid polyhedron enclosing the projection of expr on a coordinate subspace is returned; in the second case,
the sequence of measures of the edges of such cuboid is returned. For instance,

\[
\text{BOX}:(1,3):\text{expr} \quad \text{and} \quad \text{SIZE}:(1,3):\text{expr}
\]

return the (2,2) dimensional rectangle which encloses the projection of expr on the x,y coordinate subspace, and a sequence of two real numbers, respectively.

4.3 Exploiting Algebraic Identities

An algebraic calculus over polyhedral complexes is embedded within the language. The algebraic properties of such a calculus can be used by an optimizing compiler based on program transformation [Williams and Wimmers 1991] in order to derive equivalent programs to be executed more efficiently. For example, it is possible to show [Paoluzzi et al. 1993] that the skeleton extraction operation distributes over the product of cell-decomposed polyhedra, therefore requiring the computation of the skeleton of lower-dimensional cells. This is an important speedup, because the computation of the k-skeleton of a polyhedral complex requires an extensive computation of k-dimensional determinants.

For example, the two following identities hold (see also Figure 9), where $A^{2,2}$ and $B^{1,1}$ denote polyhedral complexes, the exponent index denotes the dimension of the polyhedral complex, + and * denote polyhedral union and
product, respectively, and where @k denotes the unary operation of extraction of the set of k-dimensional cells of a polyhedral complex:

$$@2: (A \star B) = (@2: A \star @0: B) + (@1: A \star @1: B)$$
$$@1: (A \star B) = (@1: A \star @0: B) + (@0: A \star @1: B)$$

Other useful algebraic properties can also be given, which may introduce important optimizations in a compiler for a geometric language. For instance, we give two other such identities. It is easy to see that the boundary operator (@: complex^d,n \to complex^{d-1,n}) and the mapping operator ((MAP:f): complex^d,n \to complex^{d,m}) commute. The same is true for any skeleton. More formally:

$$(@ \sim MAP:f): pol = (MAP:f \sim @): pol$$
$$(@k \sim MAP:f): pol = (MAP:f \sim @k): pol$$

analogously commute the CAT and STRUCT operators:

CAT \sim AA: STRUCT = STRUCT \sim CAT

Another useful algebraic law says that the product of polyhedra distributes with respect to the STRUCT function:

$$\text{STRUCT} : \langle A, B \rangle \star C = \text{STRUCT} : \langle A \star C, B \star C \rangle$$

It should be noted, with respect to the backward evaluation of polyhedral sequences executed by the STRUCT function, that the left side of the previous identity is less computationally intensive, because it is executed on a lower dimensional polyhedral sequence. The right side may be the convenient choice if lazy evaluation of geometry is enforced, so that no \* V and only the V* algorithm is executed when evaluating the STRUCT application. In any case, in order to use the previous identities as equational rewriting rules, they must be abstracted in functional form [Backus et al. 1989], where only combining forms and functions are involved.

4.4 Programming with Constraints

An example of geometric programming with a limited but valuable use of constraints is discussed in this section. We show the ease of constraint definition and use. In particular we discuss how to build a constrained plan layout, where single and grouped rooms are constrained both on their linear dimensions and on the relative positioning. In this approach, two adjacent shapes are forced to share a specified point. First, four functions, namely SW, SE, NE, and NW, are defined, which return a geographic point of an input polyhedron, as shown in Figure 10:

DEF SW = [MIN:1, MIN:2];
DEF SE = [MAX:1, MIN:2];
DEF NE = [MAX:1, MAX:2];
DEF NW = [MIN:1, MAX:2].

Then a function "^" is defined, which allows one to combine two shapes in such a way that two geographic points coincide. Such a (higher level) function is a binary composition of functions. The application of the function f ^ g to a pair of polyhedral objects \langle roomA, roomB \rangle, where both f and g are geographic

functions as defined previously, will return a structure which contains the two objects and the proper translation transformation:

```
DEF A (f,g::lsFun) = STRUCT - [s1, (T:(l, 2)) - parameters, s2]
WHERE
  parameters = (AA: - ) - TRANS - [f - S1, g - s2]
END
```

It is now possible to define 16 "geographical" binary constraint functions between two objects `roomA` and `roomB`. Four examples, graphically shown in Figure 11, follow:

```
∧ :<SE, SW>:<(roomA, roomB) ∨ :<NE, NW>:<(roomA, roomB)
∧ :<NE, SW>:<(roomA, roomB) ∨ :<SW, SE>:<(roomA, roomB)
```

Such prefix expressions are equivalent but less readable than the infix expressions which follow, where both the parenthesis and the spaces around the `∧` function are needed for syntactical reasons.

```
roomA (SE ∧ SW) roomB roomA (NE ∧ NW) roomB
roomA (NE ∧ SW) roomB roomA (SW ∧ SE) roomB
```

We are finally ready to express the parametric plan layout of Figure 12 as hierarchically constrained:

```
DEF layout (a, b, c, d, e, f, g, h::lsRealPos) =
  ((room1 (SE ∧ SW) room2)
   (NW ∧ SW)
   (room4 (NE ∧ NW) (room5 (NW ∧ SW) room7)))
   (SE ∧ SW) (room3 (NE ∧ SE) room6)
```
WHERE
room = cuboid,
room1 = room:<(a, h),
room2 = room:<(b + c, h),
room3 = room:<(d, h + g),
room4 = room:<(a + b, e + f + g),
room5 = room:<(c, f + g),
room6 = room:<(d, e + f),
room7 = room:<(c, e)
END

where room is used as an alias for cuboid. Notice that the same result could be obtained with direct use of the STRUCT semantics:

DEF layout (a, b, c, d, e, f, g, h: :lsRealPos) =
STRUCT:<(room1, T:1:a, room2, T:1:(b + c), room3, T:2:(g + h),
room6, T:<(1, 2):<:-c, -:g), room5, T:2:(f + g),
room7, T:<(1, 2):<:-:(a + b), -:(f + g), room4)
WHERE ... END

We believe that the first version is more meaningful and natural to the designer. Clearly, the geographic functions NW, SW, and so on, and the composition operator ∧ should enter the design knowledge base (i.e., some predefined PLASM package) and should be directly available to the user/designer.

Binary operators that perform hierarchical positioning and stretching can be analogously defined. First, four new geographic functions North, South, East, and West are introduced. They return the two corner points at the chosen side of the containment box of a polyhedral object.

Then a binary composition functional named “stretch” and denoted by “\ disillusioned” is defined which returns a structure when applied to pairs of functions and then applied to pairs of shapes. As previously stated, four binary operators on pairs of (hierarchical) shapes can be obtained as (North\South), (South\North), (East\West), and (West\East), if relative positioning and stretching of the second shape with respect to the first is done. If the second shape is rotated,
stretched, and translated with respect to the first, then all possible 16 binary combinations of North, South, East, and West make sense. The PLASM code which defines the four functions and the composition operator is given in Paoluzzi et al. [1994]. Such operators, possibly combined with the previous ones, allow one model easily any (parametric) plan layout definable as a hierarchical tessellation [Stiny and March 1981]. An example of variable topology is given here, which corresponds to the layout of Figure 13, where the number \( n \) of rooms in groupA depends on the length length of the corridor.

\[
\text{DEF layout}_2(\text{length}, \text{width}, c::\text{RealPos}) = (\text{corridor (South(North) groupB)} \smaller(\text{NorthWest}) (\text{closet (Northeast) groupA)})
\]

\[
\text{WHERE}
\]

\[
\text{corridor = cuboid):(\text{length, width}),}
\]

\[
\text{groupA = (QUOTE - #:n):length * QUOTE:(14),}
\]

\[
\text{n = length (INTMAX - DIV) minRoomDim},
\]

\[
\text{minRoomDim = 10.5},
\]

\[
\text{groupB = QUOTE:<24, 24> * QUOTE:(16),}
\]

\[
\text{closet = ((QUOTE:<c, c, c, c> * QUOTE:<8>) (South(North) room:<1, c>)}
\smaller(West(East) room:<7, 7})}
\]

5. EXAMPLES OF GEOMETRIC PROGRAMMING

In this section some examples of geometric programming in the architectural design domain are discussed. In particular the variational definition of some simple buildings is given. First, the definition of a dwelling as a function depending on two real parameters is shown. Second, two primitive language constructs to generate the schematic design of multifloor buildings starting from 2D plans are discussed. Finally, the power of the FL programming style is exploited to write two functions that allow the user to generate easily the geometric model corresponding to terraced housing and to an axial aggregation of several building units.
Fig. 14. Two plan models generated by instantiating with different parameter values the function plan and a model directly generated as object. (a) Model plan_1 generated by plan:\(4, 3/2\). (b) Model plan_2 generated by plan:\(5, 3/2\). (c) Different model plan_3 generated by using the MKPOL function.

5.1 Plan Definition

The plan function creates a variational 2-dimensional model of a dwelling dependent on two real parameters \(a\) and \(b\). The function generates a \((2, 2)\) polyhedral complex which contains four squared cells (rooms) of side \(a\) and a central distribution axis of width \(b\) with two extreme squared cells of side \(b\) and a central cell with sides \(b\) and \(2 \cdot (a - b)\). Such a central group of cells (corridor) is translated by \(b/2\) in the \(y\) direction with respect to the hull of the four bigger rooms (rooms). The function works properly only if \(a > b\), otherwise it generates an error.

```plaintext
DEF plan (a, b::isRealPos) = IF:<(K:ok, ID, ERROR):ThePlan
WHERE
  ok = GE:b:a,
  ThePlan = rooms (K:<a, -:b / 2> \& SW) corridor,
  rooms = QUOTE:<a, -:b, a> * QUOTE:<a, a>,
  corridor = QUOTE:<b> * QUOTE:<b, 2 * (a - b), b>
END
DEFAULT a = 4, b = 2 END
```

The complexes generated by the application of the plan function to two different sequences of arguments are bound to the names plan_1 and plan_2. The polyhedral values denoted by such names are shown in Figure 14.

```plaintext
DEF plan_1 = plan:<4, 3/2>
DEF plan_2 = plan:<5, 3/2>
```

5.2 Generation of Building Units

5.2.1 Generation by Product. The variational definition of a multifloor block building starting from the function plan is now presented. The function block constructs a \((2, 3)\) polyhedral complex, that is, a 2-dimensional model
Fig. 15. Cutaway views of four building block models generated by instantiating with different parameter values the function block.

embedded in 3-space. (See Figure 15.) The function can be instantiated with different values of the plan parameter and of the number of floors. The interfloor height is also specified when calling the function.

```
DEF block (plan::lsPolDim:(2, 2); height::lsRealPos; n_floor::lsIntPos)
  = @2:(plan * stages)
  WHERE
  stages = QUOTE(#:n_floor:height)
END
```

The body of the block function can also be more efficiently computed as (@1:plan * stages) + (plan * @0:stages), according to Section 4.3.

Some models of block buildings can be stored in the user functional environment by evaluating the following definitions. The denoted block models will be used in the examples that follow.

```
DEF block_1_3 = block:<plan_1, 3, 3>;
DEF block_2_3 = block:<plan_2, 3, 3>;
DEF block_1_4 = block:<plan_1, 3, 4>;
DEF block_2_4 = block:<plan_2, 3, 4>.
```

5.2.2 Generation by Intersection of Extrusions. A definition of the house section as a variational (2, 2) complex of polyhedra is now given. The section, shown in Figure 16(a), is composed of three convex polyhedra. The first floor has width a + b, and the two polyhedra at the second floor have width a and b, respectively. All the polyhedra have external height h.
Fig. 16. (a) The house section generated by Section:\((4 + (3/2), 4, 3)\). Cutaway views of two \((2, 3)\) models generated by House_1 (b), House_2 (c), respectively.

DEF section \((a, b, h::\text{lsRealPos})\)
\[
= \text{STRUCT}:(\text{firstFloor}, T:2:h, \text{secondLeft}, T:1:a, \text{secondRight})
\]
WHERE
\[
\text{firstFloor} = QUOTE:(a + b) * QUOTE:(h),
\]
\[
\text{secondLeft} = \text{MKPOL}:((((O, O), (a, O), (a, h*5/4), (O, h)), (1..4), ((1))),)
\]
\[
\text{secondRight} = \text{MKPOL}:((((0, 0), (b, 0), (b, h), (0, h*6/4)), (1..4), ((1))),)
\]
END

A user-defined House function to generate models with variable plan and section is given, together with examples of the application of the function on a suitable pair of polyhedrally typed \((2, 2)\) dimensional actual parameters. The user-defined generating function is:

DEF House (Plan, Section::lsPolDim: (2, 2))
\[
= (@2 - &&:((1, 2, O), (1, O, 2))): (\text{Plan, Section})
\]
where the intersection of extrusions function \&\& is applied to a pair of sequences of integers in order to specify how to embed the 2D arguments from the space \(x, y\) into the space \(x', y', z'\) of the 3D result. A more readable alternative for the body of the House function is the infix form:

\[
@2:(\text{Plan ((1,2, O) \&\& (1, O, 2)) Section})
\]

The algorithm for the computation of the intersection of extrusions \&\&, considered as a special case of a more general product of cell complexes, is given in Bernardini et al. [1993]. In this case \&\&:\((1, 2, 0), (1, 0, 2))::(\text{Plan, Section})\] means that the Plan formal parameter is embedded in the subspace \(z' = 0\) of \(\mathbb{R}^3\) so that its \(x, y\) axis is identified with \(x', y'\) in \(\mathbb{R}^3\). Analogously, the Section parameter is embedded in the subspace \(y' = 0\) of \(\mathbb{R}^3\) so that its \(x, y\) axis is identified with \(x', z'\). The models displayed in Figures 16(b) and 16(c) are then generated by evaluating House_1 and House_2, where:

DEF House_1 = House:\((\text{plan}_1, \text{Section}: (4 + (3/2), 3))\)
DEF House_2 = House:\((\text{plan}_2, \text{Section}: (5, 5 + (3/2), 3))\)

A more interesting variational model of such houses, depending only on the measures \(a, b,\) and \(h\) of the room side, the corridor, and the interfloor height,
respectively, can be obtained by simply linking the values of parameters of the functions plan and section. First we give an object of this kind:

\[
\text{DEF HouseObj} = \text{House}:(\text{plan}: (4, 3/2), \text{Section}: (4, 4 + (3/2), 3))
\]

then by abstraction we define a function depending on such parameters:

\[
\text{DEF HouseFun} (a, b, h::\text{isRealPos}) = \text{House}:(\text{plan}: (a, b), \text{Section}: (a, a + b, h))
\]

A definition as elementary object (0-ary function) is now given for the 2D building section shown in Figure 17(a).

\[
\text{DEF Section}_1 = \text{MKPOL}:(\text{Vertices}, \text{ConvexCells}, \text{PolCells})
\]

WHERE
\[
\text{Vertices} = \langle(0,0), (0,3), (3,3), (3,0), (9,3), (9,0), (0,6), (3,6), (9/2,15/2), (9,6)\rangle,
\]

\[
\text{ConvexCells} = \langle(1,2,3,4), (3,4,5,6), (2,3,7,8), (3,8,9,10,5)\rangle,
\]

\[
\text{PolCells} = \langle(1), (2), (3), (4)\rangle
\]

The house section Section _1 given previously can be combined with the plan plan _3 generated by using a MKPOL function [see Figure 14(c)] in order to obtain a (2,3) model, which gives an example of variational geometry with varying topology obtained by using the House function:

\[
\text{DEF house}_3 = \text{House}:(\text{plan}_3, \text{section}_1)
\]

5.3 Aggregations of Building Units

5.3.1 Simple Terraces. In order to generate an aggregation of spatial units some very simple new PLASM functions can be defined, which implement the aggregation rules imposed by the architect. For instance, if several building units (e.g., houses) must be close to each other along one side, giving a terrace aggregation, the primitive function RIGHT can be used, so that each successive house is put on the right side of the previous one:

\[
\text{DEF Terrace} (\text{times}::\text{isIntPos}; \text{buildingUnit}::\text{isPol}) = \text{INSR}:\text{RIGHT}:(\#\text{times}; \text{buildingUnit})
\]

where buildingUnit is any model defined in the previous section. It may be useful to see how an application of the function Terrace is reduced on actual
parameters. For example, if the terrace aggregation of 4 instances of the object pol is desired, we have:

\[
\text{Terrace}(4, \text{pol}) \\
= \text{INSR:RIGHT} : \{\#4: \text{pol}\} \\
= \text{INSR:RIGHT} : \langle \text{pol}, \text{pol}, \text{pol}, \text{pol} \rangle \\
= \text{RIGHT} : \langle \text{pol}, \text{RIGHT} : \langle \text{pol}, \text{pol} \rangle \rangle
\]

Notice that the \text{RIGHT} predefined function works as follows:

\[
\text{DEF} \quad \text{RIGHT} (\text{pol}1, \text{pol}2 : \text{IsPol}) = \text{STRUCT} : \langle \text{pol}1, \text{translation}, \text{pol}2 \rangle \\
\text{WHERE} \\
\quad \text{translation} = T : 1 : \langle \text{MAX} : 1 : \text{pol}1, \text{MIN} : 1 : \text{pol}2 \rangle \\
\text{END}
\]

Some terraced aggregations of several houses defined in the preceding are given in Figure 18. They are generated, respectively, as follows:

\[
\text{DEF} \quad \text{terrace}_1 = \text{Terrace} : \langle 5, \text{HouseFun} : \langle 4, 3 / 2, 3 \rangle \rangle; \\
\text{DEF} \quad \text{terrace}_2 = \text{Terrace} : \langle 6, \text{block} \_1 \_4 \rangle; \\
\text{DEF} \quad \text{terrace}_3 = \text{Terrace} : \langle 7, \text{house} \_3 \rangle
\]

We would like to explicitly note that the \text{Terrace} function is a representation of a class of objects with varying topology and shape.

5.3.2 Axial Aggregation. A different aggregation rule, implementing an axial composition along the direction of the \(x\) axis, is given by the following function \text{AxialComp}, where \text{ALIGN} is a primitive operator described in Paoluzzi and Sansoni [1992].
Fig. 19. Some axial aggregations of building units. (a) axial_1; (b) axial_2; (c) axial_3.

DEF AxialComp (buildings::isSeqOf:isPol) = INS:axialAlignment:buildings
WHERE
  axialAlignment = ALIGN:((1, MAX, MIN), (2, MED, MED))
END

Some axial aggregations of several building units previously defined are
given in Figure 19. They are generated, respectively, as

DEF axial_1 = AxialComp:(block_1_3, block_2_4, block_1_4, block_2_3);
DEF axial_2 = (AxialComp ~ ##:2 ~ AA:HouseFun):(4, 3/2, 3), (5, 3/2, 3));
DEF axial_3 = (AxialComp ~ ##:3):(house_1, house_3).

6. EXPERIENCE AND FUTURE DIRECTIONS

The interaction of the language interpreter with the user-functional environ-
ment is described briefly in this section, in order to outline some main choices
in our language implementation. The interpretation of a function definition
always modifies the user-functional environment. The evaluation of a lan-
guage expression may also sometimes modify such an environment, which
can be considered a sort of design base. Both global and local functions are
associated with a tuple in the functional environment. When a function
definition is interpreted, the user-functional environment is updated by
adding new tuples and possibly modifying some references in other tuples.

A function defined using formal parameters needs to be applied on actual
parameter values. The value returned by the application clearly depends on
such values. Conversely, if a function was defined without formal parameters
its Body always denotes the same value. Such a value is stored when first
computed and simply returned at any subsequent request. We call such functions *value oriented*. The value validity is implicitly checked during the whole work session. In fact, a redefinition of any function used to compute it makes such a value invalid. It is a good programming practice to use a value-oriented function when a value is repeatedly used inside the session and when it is derived with an expensive computation. When the user needs to change some previous design decisions, he or she has only to redefine some functions. From a geometric computing point of view, a change in a component will imply the recomputation of the only parts of the shape that are actually influenced by the updated component.

The simple evaluation strategy we have described helps to save time and storage when the same object is referenced more than once. Furthermore, if some object is never used (a node is never reached) in evaluating a top-level expression, no useless effort is made to compute it.

The reader might like to read some information about the state of our project for the development of the design language PLASM. A dimension-independent geometric modeler based on both face- and vertex-based decompositions in convex cells, starting on the ground of the *Simple* prototype modeler [Paoluzzi 1993], has been in development since late 1992. A homemade interpreter for both PLASM and an FL subset was developed in early 1993. It is currently working both on Unix workstations and on Apple Quadras running Common Lisp. Starting in early 1993 the CAD group also developed a lower-level imperative modeling and simulation language oriented to robotics application, called RobLan, where dimension independence is limited to the range 0D–3D. Although most of what is described in this article is currently running in PLASM, this is not yet true for Boolean unions and differences of polyhedral complexes, and consequently for the backward evaluation of polyhedral sequences based on progressive difference. More research and development effort is needed in order to implement efficient and reliable dimension-independent set operations. Currently, we can automatically translate an internal multigraph representation of polyhedral sequences with fixed low dimension (≤ 3) in a RobLan program which may execute the backward evaluation algorithm, thus producing a valid representation as polyhedral complex.

Our long term goal is to develop a visual language shell around a PLASM nucleus. Thus, it will become possible to build personalized graphics interfaces where both the designer/user and the PLASM application programmer will be comfortably accommodated. In the interim we plan to (a) complete a theory about our HPC “weak representation” [Pascucci 1994] of polyhedral complexes; (b) implement dimension-independent Booleans using the HPC representation with Nef's polyhedra [Bieri 1994] as primitive polyhedra, in turn represented by using the Thibault and Naylor BSP approach [Foley et al. 1990; Naylor 1990; Thibault and Naylor 1987]; (c) develop a PLASM implementation of some generative graphics described in Snyder [1992], in order to both test the language and enrich it with powerful new primitives; (d) implement both PLASM and the translator towards RobLan by using the optimizing compiler for FL; (e) extend the language with some explicit

equational mechanism for constraint definition and solving. Some PLASM operators, including the dimension-independent offset operation ++, also require more research and/or a more efficient implementation.

7. CONCLUSION

This article has discussed a programming approach to geometric design based on functional programming. The main characteristics of such an approach are a dimension-independent view of geometric modeling and a programming implementation of variational geometry.

As the examples of Section 5 have shown, any PLASM function can be considered as a "generating form" which is able to produce, depending on the value of actual parameters, infinitely different geometric models with some common structure. This goes beyond the standard variational geometry approach, where the instances from a family of constrained shapes must all have the same topology and where shape instances differ only by the numeric values of shape parameters. Dimension independence allows for writing code that can be applied to geometric objects of any dimension. Other possibilities of dimension independence concern (higher-level) parameterized shapes and modeling of objects with material properties varying both spatially and in the time dimension.

We also note that with a programming approach the design decisions are made completely explicit, so that they can be compactly stored on electronic media and easily transmitted on communication lines, as well as recognized and updated in subsequent steps of design review. Such an approach to design is consistent with the current research on collaborative and distributed design [Godse 1991] and with the recent developments in transfer formats for CAD/CAM and product data. Such an industrial trend is mainly represented by the language EXPRESS [Iso Technical Committee 1992] defined within the STEP ISO standard.

In conclusion, the authors believe that this article and the current prototype interpreter may show that the described functional approach can be very useful for geometric modeling. In particular, a geometric programming approach based on the functional paradigm seems to have an amazing descriptive power. For instance, it has been used [Paoluzzi 1994] to implement with few lines of code d-dimensional Bezier manifolds of any degree, as well as GIS applications and CAD operations (e.g., profile products). Also, in order to describe and manipulate motions, a language for robot programming and graphical simulation would require both some specialized data types (e.g., matrices with symbolic elements) and the implementation of a subsystem for symbolic manipulation. It is possible to see that such tools are given for free in a PLASM environment.

APPENDIX

Glossary

An alphabetic glossary of some predefined PLASM functions is given in this Appendix, in order to make the examples discussed in this paper easier to
understand. A more complete list of predefined functions and predicates can be found in Vicentino [1994].

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function application</td>
<td>+:(1,3) = 4</td>
<td></td>
</tr>
<tr>
<td>composition operator</td>
<td>(f ∘ g):x = f.(g:x)</td>
<td></td>
</tr>
<tr>
<td>infix sequence constructor</td>
<td>2.5 = (2, 3, 4, 5)</td>
<td></td>
</tr>
<tr>
<td>binary product of polyhedra</td>
<td>pol1 * pol2</td>
<td></td>
</tr>
<tr>
<td>product of numbers</td>
<td>2 * 4 = 8</td>
<td></td>
</tr>
</tbody>
</table>
| “raised” product of functions | (sin * cos):x = 
MUL: <sin(x), cos(x)> |
| product of matrices | <(1, 2), (2, 4)> * <(5, 6), (7, 8)> = <(19, 22), (38, 44)> |
| binary difference of polyhedra | pol1 - pol2 |
| “raised” subtraction of numbers | -:(3, -8) = 11 |
| “raised” division of numbers | /:(5, 2) = 2.5 |
| repetition operator | #:3:expr = (expr, expr, expr) |
| sequence repetition operator | ##:3:(a, b) = (a, b, a, b, a, b) |
| binary intersection of polyhedra | pol1 & pol2 |
| binary intersection of extrusions | &:<((1, 2, 0), (1, 0, 2))>:<plan, sec> = plan((1, 2, 0) & & (1, 0, 2)) sec |
| sum of no. and union of pol. | pol1 + pol2 |
| offset of polyhedra | + + :pol^n - 1.n:3 = pol^n |
| boundary operator | @:pol |
| k-skeleton extractor | @:2:pol |
| apply-to-all | AA:f:<x1,...,xn> = <f:x1,...,f:xn> |
| absolute value | ABS: – 8 = 8 |
| apply-in-composition | AC:SEL:{2, 1} = SEL:2 ∼ SEL:1 |
| append left | AL:<x, <y1,...,yn>> = <x, y1,...,yn> |
| relative positioning operator | ALIGN:<(1, MAX, MIN), (2, MED, MED)> |
| apply-in-sequence | AS:SEL:{2, 1} = [SEL:2, SEL:1] |
| containment box extractor | BOX:<(1, 2):pol |
| catenate | CAT:<(a, b, c),..., (x, y, z)>> = 
(a, b, c,..., x, y, z) |
| composition | COMP:{f1, f2,..., fn} = f1 ~ f2 ∼ ... ∼ fn |
| construction | CONS:{f1,..., fn}:x = 
[f1:x,..., fn:x] |
| cuboid constructor | CUBOID:<x1,..., x6> |
| keyword for default definitions | DEFAULT f1 = expr1, f2 = expr2 END |
| keyword for function definition | DEF f (a, b)(x, y):typePred = expr |
| keyword for default values | (S:<(1, 2, 3):<(r, r, R)>):(cylinder:DF) |
| intrinsic dimension predicate | DIM:pol |
| distribute left | DISTL:<x, <a, b, c>> = 
<x, a>, <x, b>, <x, c>> |
| distribute right | DISTR:<(a, b, c), x> = 
<a, x>, <b, x>, <c, x>> |
<p>| embedding operator | EMBED:mpol^n - n = pol^n-m |
| keyword for local definitions | END WHERE f1 = expr1, f2 = expr2 END |
| truth value | FALSE |
| sequence constructor | FROMTO:&lt;(2, 5), &lt;2, 3, 4, 5)&gt; |
| greater or equal | GE:2:2 = TRUE |
| greater than | GT:8:2 = FALSE |</p>
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>elementary shearing</td>
<td>H:3:(h_x, h_y, 1)</td>
</tr>
<tr>
<td>ID</td>
<td>identity</td>
<td>ID:x = x</td>
</tr>
<tr>
<td>IF</td>
<td>conditional form</td>
<td>IF:&lt;(p, f), g&gt;:x</td>
</tr>
<tr>
<td>INS L</td>
<td>insert left</td>
<td>INS L:f:(x_1,... ,x_n) = f:(INS L:f:(x_1,... ,x_{n-1}), x_n)</td>
</tr>
<tr>
<td>INS R</td>
<td>insert right</td>
<td>INS R:f:(x_1,... ,x_n) = f:(x_1, INS R:f:(x_2,... ,x_n))</td>
</tr>
<tr>
<td>INT MAX</td>
<td>smallest higher integer</td>
<td>INT MAX:3.5 = 4</td>
</tr>
<tr>
<td>INT MIN</td>
<td>greatest lower integer</td>
<td>INT MIN:3.5 = 3</td>
</tr>
<tr>
<td>INV</td>
<td>matrix inversion</td>
<td>INV:&lt;(1, 1), (1, 0) = &lt;(0, 1), (1, -1)&gt;</td>
</tr>
<tr>
<td>IS INT</td>
<td>test pred. for integers</td>
<td>IS INT: -5 = TRUE</td>
</tr>
<tr>
<td>IS INT Pos</td>
<td>test pred. for pos. int.</td>
<td>IS INT Pos: -5 = FALSE</td>
</tr>
<tr>
<td>IS FUN</td>
<td>test pred. for functions</td>
<td>IS FUN: CAT = TRUE</td>
</tr>
<tr>
<td>IS NULL</td>
<td>test the empty sequence</td>
<td>IS NULL: &lt;{} = TRUE</td>
</tr>
<tr>
<td>IS POL</td>
<td>test pred. for polyhedra</td>
<td>IS POL:QUOTE:&lt;(5)) = TRUE</td>
</tr>
<tr>
<td>IS POL Dim</td>
<td>test pred. for dim. of pol.</td>
<td>IS POL Dim:&lt;(1, 1)]:QUOTE:&lt;(5)) = TRUE</td>
</tr>
<tr>
<td>IS REAL</td>
<td>test pred. for reals</td>
<td>IS REAL: -5.2 = TRUE</td>
</tr>
<tr>
<td>IS REAL Pos</td>
<td>test pred. for pos. reals</td>
<td>IS REAL Pos: -5.2 = FALSE</td>
</tr>
<tr>
<td>IS SEQ</td>
<td>test pred. for sequences</td>
<td>IS SEQ:&lt;(a, b, c) = TRUE</td>
</tr>
<tr>
<td>JOIN</td>
<td>join operator</td>
<td>JOIN:&lt;(p01, p02, p03)</td>
</tr>
<tr>
<td>K</td>
<td>constant function</td>
<td>K: x_1:x_2 = x_1</td>
</tr>
<tr>
<td>LEN</td>
<td>sequence length</td>
<td>LEN:&lt;(a, b, c) = 3</td>
</tr>
<tr>
<td>MAP</td>
<td>map operator</td>
<td>MAP: f:pol</td>
</tr>
<tr>
<td>MAT</td>
<td>linear mapping constructor</td>
<td>MAT:&lt;(a_{11}, a_{12}, a_{13}), (a_{21}, a_{22}, a_{23})</td>
</tr>
<tr>
<td>MAX</td>
<td>maximum coordinate extractor</td>
<td>MAX: i:pol</td>
</tr>
<tr>
<td>MIN</td>
<td>minimum coordinate extractor</td>
<td>MIN: i:pol</td>
</tr>
<tr>
<td>MK POL</td>
<td>polyhedron constructor</td>
<td>MK POL:&lt;(VERTS, CONV CELLS, POL CELLS) = pol</td>
</tr>
<tr>
<td>MUL</td>
<td>binary multiplication of numbers</td>
<td>MUL:&lt;(3, 5) = 15</td>
</tr>
<tr>
<td>PI</td>
<td>(\pi) constant</td>
<td>2*PI = 6.2831...</td>
</tr>
<tr>
<td>QUOTE</td>
<td>1D polyhedron constructor</td>
<td>QUOTE:&lt;(x_1,... ,x_n)</td>
</tr>
<tr>
<td>R</td>
<td>elementary rotation</td>
<td>R:&lt;(1, 2)&gt;:alpha</td>
</tr>
<tr>
<td>REVERSE</td>
<td>reverse a sequence</td>
<td>REVERSE:&lt;(y_1,... ,y_n) = (y_n,... ,y_1)</td>
</tr>
<tr>
<td>RIGHT</td>
<td>right alignment of polyhedra</td>
<td>RIGHT:&lt;(p01, p02)</td>
</tr>
<tr>
<td>RN</td>
<td>embedding dimension predicate</td>
<td>RN:pol</td>
</tr>
<tr>
<td>S</td>
<td>elementary scaling</td>
<td>S:&lt;(1, 2, 3)&gt;:s_x, s_y, s_z</td>
</tr>
<tr>
<td>S1, S2, ...</td>
<td>specified selector operator</td>
<td>[s_2, s_3]:&lt;(a, b, c, d) = (b, c)</td>
</tr>
<tr>
<td>SEL</td>
<td>generic selector operator</td>
<td>SEL:2:&lt;(a, b, c, d) = b</td>
</tr>
<tr>
<td>SIGN</td>
<td>sign extractor</td>
<td>SIGN:3 = 1; SIGN:0 = 0</td>
</tr>
<tr>
<td>SIZE</td>
<td>size extractor</td>
<td>SIZE: i:pol</td>
</tr>
<tr>
<td>SIMPLEX</td>
<td>simplex constructor</td>
<td>SIMPLEX:3</td>
</tr>
<tr>
<td>STRUCT</td>
<td>structure operator</td>
<td>STRUCT:&lt;(p01, T:1:a, p02)</td>
</tr>
<tr>
<td>TAIL</td>
<td>tail extractor of a sequence</td>
<td>TAIL:&lt;(x_1,... ,x_n) = (x_2,... ,x_n)</td>
</tr>
<tr>
<td>TRANS</td>
<td>transpose a sequence of sequences</td>
<td>TRANS:&lt;(a, b, c), (x, y, z) = (x, y, z), (a, b, c)</td>
</tr>
<tr>
<td>TRUE</td>
<td>truth value</td>
<td>T:&lt;(1, 2, 3):t_x, t_y, t_z</td>
</tr>
<tr>
<td>T</td>
<td>elementary translation</td>
<td>T:2:1</td>
</tr>
<tr>
<td>UK POL</td>
<td>&quot;inverse&quot; of MK POL</td>
<td>UK POL:pol=&lt;(VERTS, CONV CELLS, POL CELLS)</td>
</tr>
<tr>
<td>WHERE</td>
<td>keyword for local definitions</td>
<td>WHERE 11 = expr 1, I2 = expr 2 END</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

This paper is dedicated to Gianfranco Carrara, who several years ago advised one of the authors to explore the field of languages in order to model the geometry of design. We are strongly indebted to the referees and the editor because their criticism, comments, and suggestions were of great help in improving the presentation of this article and better highlighting the main ideas underlying our work. We would also like to thank Alberto Pettorossi for useful discussions about typing and interpretation of functional languages and Peter Lucas for many comments on a previous paper. His suggestions about a better use of FL have oriented the implementation of our prototype interpreter for PLASM. Thanks also to John Williams for the FL optimizing compiler and to Charles Eastman for good advice about the use of spatial indices in Boolean operations.

REFERENCES


Geometric Programming  .  305


SYELEE, G. L. Jr. 1980. The definition and implementation of a computer programming language based on constraints. Stanford University, Stanford, Calif., Ph.D. Thesis. (Published as MIT-AI TR 595.)


Received March 1995; accepted July 1995

Editor Andrew S. Glassner