PROGRAMMING IN THREE DIMENSIONS

BY

MARC-ALEXANDER NAJORK

Dipl.-Wirtsch. Inf., Technische Hochschule Darmstadt, Germany, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

PROGRAMMING IN THREE DIMENSIONS

Marc-Alexander Najork, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
Simon M. Kaplan, Advisor

This thesis describes Cube, the first visual language to employ a three-dimensional syntax. The third dimension provides for a richer syntax, makes the language more expressive, and prepares the ground for novel, virtual-reality-based programming environments. We use dimensional extent to convey semantic meaning, or more precisely, to distinguish between logical disjunctions and conjunctions, and between sum and product types.

Cube uses the data flow metaphor as an intuitive way to describe logic programs. The semantics of the language is based on a higher-order form of Horn logic. Predicates are viewed as a special kind of terms, and are treated as first-class values. In particular, they can be passed as arguments to other predicates, which allows us to define higher-order predicates.

Cube has a static polymorphic type system, and uses the Hindley-Milner algorithm to perform type inference. Well-typed programs are guaranteed to be type-safe.

We have implemented two Cube interpreters: An initial feasibility study, and a prototype implementation with improved interactive capabilities. Both of them exploit the implicit parallelism of the language by simulated concurrency, implemented via time-slicing.

To my family

ACKNOWLEDGEMENTS

The design of Cube went through a number of iterations before it reached its final form. It was shaped by continuous discussions with two people: my adviser, Prof. Simon Kaplan, and Prof. Eric Golin, who introduced me to visual programming. I am deeply grateful to them for their advice and their friendship.

I am equally grateful to Prof. Sam Kamin and Prof. Uday Reddy, who taught me most of what I know about functional programming and about programming language semantics. Their course on the implementation of lazy functional languages proved to be particularly useful. Many of the implementation details of the CUBE-I system can be traced back to this class. Their weekly "theory of programming" seminar was a great source of stimulation to me throughout the years.

I am also grateful to Prof. Laxmikant Kalé and Prof. Gul Agha, who both taught me a great deal about concurrent programming and the potential pitfalls.

I wish to thank Prof. Ralph Johnson for exposing me to the whole spectrum of object-oriented programming, from Smalltalk to C++. The fact that the CUBE-II system is implemented in an object-oriented language is tribute to his convincing and persuasive arguments in favor of object-oriented languages.

With around 400 graduate students, the Computer Science department at the U of I is the largest in the nation. Many of these people have been good friends to me, and I wish to thank them all collectively. In particular, I would like to thank T. K. Lakshman, Jonathan Springer, Rune Dahl, and Josh Caplan for the many insights they have provided during our discussions on language semantics and implementation, and for being great pals. Thanks also to Elsa Bignoli, Doug Bogia, the unforgettable Alan Carroll, Mark Kendrat, Ted Phelps, Bill Tolone, and Esmeralda Wijngaarde.

A very special "thank you!" goes to Riccardo Bettati, for being the best friend (and best man!) that he has been. I am glad that we are both moving to the same corner of the world!

During the summer of 1992, I performed an internship at Digital Equipment's Systems Research Center. I think this summer was one of the most productive periods of my life. I wish to thank Marc Brown for having been a great host and mentor as well as being a good friend. Stephen Harrison, who became an impromtu second host to me during this time, deserves similar praise. And thanks to Bob Taylor who demonstrated confidence in me by allowing me to join SRC for good. I am very much looking forward to work together with all the SRCers.

The research assistantship which carried me through much of my tenure as a graduate student was paid through NSF grant CCR-9007195. I wish to thank the National Science Foundation for this support.

Nothing of all this would have been possible without my parents. They have helped me in every respect, by giving me a good home, supporting me through most of my life, and encouraging me in all my endeavors.

Finally, I am grateful to my lovely wife, Ebtisam Abbasi, for cheering me up when my spirits are low, keeping up with me when I am a pain (which is more often than not), and giving perspective to my life.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Programming is the activity of describing an algorithm in a formal notation — a programming language — for the purpose of then executing it on a computer. The art of programming goes back a long time, among its first practitioners were such pioneers as Charles Babbage and Augusta Ada Lovelace. However, programming became more widespread only with the advent of the electronic digital computer in the 1940's. The very first computer programs were formulated in very low-level languages, which strongly reflected the underlying machine architecture. However, within a decade higher-level, "problem-oriented" programming languages came into existence, with Fortran being the patriarch of a long line.

The vast majority of these languages are *textual* in nature. That is, phrases in these languages are formed from (essentially linear) strings of characters of some alphabet (a notable exception is Zuse's Plankalkül [88], a language devised in 1945 for one of the first computers, which uses a two-dimensional layout for textual formulas). This limitation to textual languages reflected to some extent the limitations of the available input/output hardware (teletype keyboards and typewriters). It also reflected the fact that textual notations are fairly easy to parse, and thus made efficient use of the then scarce computing resources.

Computer graphics hardware has become available in the 1960's. The year 1963 marks the creation of the first graphical application program, Ivan Sutherland's Sketchpad [82]. Three years later, his brother William created what might be considered the first *visual programming language* [84]. The system was based on a data flow metaphor; it allowed its user to select

operators from a menu and to connect them using a light pen, to assign input values to the "circuit", and to watch it execute, all in real time.

The 1960's saw the creation of two other visual programming languages. The AMBIT family of languages is based on two-dimensional pattern-matching and diagram rewriting. AMBIT/L [16] was designed for rewriting graphical representations of list structures, AMBIT/G [15] is aimed at general graphs. The GRAIL language [21], developed at RAND, uses a flowchart notation to specify programs visually.

Since then, computer graphics hardware became more and more affordable. In the early 1980's, bit-mapped displays and pointing devices such as mice were first integrated into personal computers; today, they are standard equipment of base-level systems. Parallel to this development of hardware technology, a multitude of visual programming languages have been devised ([10, 19, 25, 77] being some of the earlier ones). Although they have not left a mark on general-purpose programming, they have carved a niche in specialized application domains, such as providing simple programming interfaces to end-users of scientific visualization systems (examples are Iris Explorer [76], AVS [87], and apE [61]).

Ivan Sutherland also pioneered a technology that has since then become known as "Virtual Reality". He designed a head-mounted display that allowed its user to view three-dimensional, computer-generated images in a stereoscopic fashion [83, 85]. For the next 25 years, this technology remained too expensive to become widespread. Today, however, 3D graphics hardware is becoming more and more affordable; at the same time, novel input/output devices, such as head-mounted displays and data gloves, which allow a user to immerse himself into a computer-generated 3D visualization and to directly interact with objects in this scene, are becoming commercially available.

Will these advances in input/output hardware lead to a new family of visual programming languages? How feasible are those languages, and what are their benefits over 2D visual and over textual languages? These are the questions that have motivated this thesis.

1.1 Visual Languages

A visual language is a programming language which uses a predominantly graphical notation. The field can be traced back to work done by William Sutherland [84] and others in the mid

1960's; however, most of the research has been done within the last 10 years, when low-cost computers with high-resolution bit-mapped graphics capabilities became widely available.

A great number of different visual paradigms have been explored. A detailed survey of the various approaches is beyond the scope of this thesis. We just want to describe briefly three major paradigms: *control flow*, *data flow*, and *visual rewriting*. For a more detailed survey of visual languages, the reader is referred to compendia such as [11, 75].

The *control flow paradigm* uses the flow chart metaphor to describe the control flow of a program. Simple operations, such as assignment or procedure invocation, are depicted as boxes; sequencing is denoted by arrows connecting two boxes. In addition, there are visual representations for the common control constructs, such as conditionals or iteration. So, control flow language are based on the same semantic model as textual procedural languages of the Fortran- and Algol-families. Pict [25] is an archetypical control flow language.

The *data flow paradigm* uses boxes to denote functions, and arrows to connect the output of one function to the input of another. Data flow languages are typically stateless, and are based on the semantic model used by functional languages. However, almost all of them are first-order (refer to Section 2.3 for a discussion of the exceptions), thereby giving up much of the power of functional languages. Show-and-Tell [37] is a typical data flow language.

The *visual rewriting paradigm* employs some form of visual rewrite rules to describe how a given picture shall be transformed into another picture. Languages belonging to this group are very diverse in their appearance, although they are all based on the same idea: match a picture or a part of a picture against the left-hand side of a rewrite rule, and replace it by the rule's right-hand side. Christensen's AMBIT languages [15, 16] operate on graph- and on list-structures, Furnas' BITPICT system [23] operates on pixel-arrays, and Kahn and Saraswat's Pictorial Janus [35] rewrites closed contours.

1.2 Strengths of Visual Languages

There are many arguments in favor of visual programming. Usually, these arguments center around the fact that humans are known to process pictures easier and faster than text. According to Raeder,

> "It is commonly acknowledged that the human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text." (cf. [67] page 12)

Raeder then outlines more specifically why humans can cope with pictures better than with text:

- **Random vs. Sequential Access:** Text is of a sequential nature, while pictures provide random access to any part, as well as detailed and overall views.
- **Transfer Rate:** As the human sensory system is set up for real-time image processing, pictures can be accessed and decoded more rapidly.
- **Dimensions of Expressions:** Text is one-dimensional in nature, while pictures are multi-dimensional, and provide through visual properties like color, shape, size, and direction a richer language, which can lead to a more compact encoding of information.
- **Concrete vs. Abstract:** Pictures can provide concrete metaphors, which make it easier to grasp an abstract idea.

Raeder's justification of visual programming is largely psychological in nature, and can be verified only through empirical studies. The basic premise, however, is quite plausible. Programmers commonly use pictures to develop algorithms or data structures and to communicate them to other programmers.

There have been several attempts to measure the benefits of visual languages and of program visualization systems. Pandey and Burnett [62] conducted an empirical comparison between Forms/3 [7] (a form-based visual language), Pascal, and APL. They investigated the usefulness of these languages in a very narrow problem domain, namely matrix multiplication. In particular, they tested how well the test subjects could solve two problems with each language: appending two matrices of compatible size, and computing the first n Fibonacci numbers.

They conducted this test on 60 students. All of the students had prior experience with Pascal or C, one had experience with APL, and none had prior experience with Forms/3. The test subjects were given a 40-minute lecture on the application of these three languages to various matrix-manipulation tasks. Afterwards, the students had to solve each of the two problems in

Results for Problem 1: Appending two matrices				
	completely correct	nearly correct	conceptually but not logically correct	incorrect
Pascal	7	1	21	31
Forms/3	53	0	2	5
APL	49	3	2	6
Results for Problem 2: Computing Fibonacci numbers				
	completely correct	nearly correct	conceptually but not logically correct	incorrect
Pascal	38	5	4	13
Forms/3	35	9	7	9
APL	15	3	6	36

Table 1.1: Results of Pandey's and Burnett's User Study

each of the three languages, that is, they had to write 6 programs. They were allowed 5 minutes per program. The order of languages in which the solutions had to be constructed was varied among the participants.

Table 1.1 shows the outcome of the experiment. For the first problem, appending two matrices, Forms/3 and APL were superior to Pascal, while for the second problem, computing the first n Fibonacci numbers, Forms/3 and Pascal outperformed APL.

The application domain on which this study focused, namely matrix multiplication, is admittedly very narrow. Furthermore, one can of course not generalize from a particular visual language to the whole genre. However, this study at least suggests that for certain tasks, an appropriate visual language has the potential to outperform its textual competitors.

1.3 Weaknesses of Visual Languages

While visual programming *per se* shows a great deal of potential, there are also a lot of problems associated with existing visual languages. Some of them are related to the visual notation they employ, while others reflect trends in language design that have dominated the visual language community up to now. We highlight four of those problems below:

- The screen space problem

- Low execution speed
- The lack of static type systems
- Conservative semantics

Visual languages tend to use a relatively sparse notation, that is, they use more screen real estate to describe a given problem than textual languages do. This problem is usually called the *screen space problem*. It can be alleviated by use of procedural abstraction, i.e. by collapsing subdiagrams into single symbols, and treating these symbols as "black boxes" similar to predefined operators.

Most visual languages are interpreted, and in many cases, the interpreter operates on a representation that is very close to the picture representation of the program. This results in poor performance of visual programs, and has given them a reputation of being inefficient.

The vast majority of existing visual languages are latently typed, which means that they check whether a given operator receives values of the appropriate type only when this operator is actually applied, i.e. at run time. One reason for this is that much of the research in visual languages has focused on developing languages for non-programmers, and that the notion of types was considered to be too complicated. However, not verifying the well-typedness of a program at compile-time means that type errors are discovered only by trial-and-error. In addition, latently typed languages typically use some form of run-time type checking, which leads to significant execution overheads.

Many visual languages are based on relatively traditional semantic models. Those languages whose syntax is based on the control flow paradigm are semantically procedural languages, i.e. descendents of Fortran or Algol. As the name suggests, the notation focuses on control structures that manipulate data instead of the data itself. The emphasis is on "how to do it" rather than "what to do". The level of abstraction in these languages is low compared to other paradigms, and programming mistakes are easily made.

Languages whose syntax is based on the data flow paradigm are by and large functional languages. The functional framework is one of the cleanest and most elegant language paradigms, although it is hard to build efficient implementations. One of the most powerful features of modern functional languages is that they treat values as functions, i.e. as ordinary data, which can be passed around as arguments to other functions. Functions which take other functions

as arguments are called higher-order, they allow for powerful abstractions, and their liberal use can greatly increase the potential for code reuse. Strangely enough, though, almost all visual languages are first-order, and thereby miss out on this powerful abstraction mechanism. In fact, to our best knowledge, we were the first to propose the use of higher-order functions in the data flow framework [56].

1.4 Arguments for a 3D Notation

Six years ago, Ephraim Glinert speculated about the prospects and potential benefits of three-dimensional visual programming:

> "But first, why do we advocate programming in three dimensions? Many readers will surely argue ..., that we don't yet know how to properly utilize two dimensions!
>
> We do not propose eschewing 2-D visual and iconic programming for 3-D. We *do* propose broadening our horizons to include the third dimension when appropriate, for several reasons. For one reason, the technology is now available in (top of the line) workstations, and it will rapidly become affordable to all. Most importantly, however, are the precedents set by analogy with other branches of science. ...
>
> Might not a similar situation hold for programming, too? And, if it does, ought we not to exploit it to our advantage? ... It is time for computer science to begin exploring revolutionary rather than evolutionary means of programming, in the hope that the tools will be ready when required." [24]

In response to Glinert's call, we have developed Cube [57, 58], the first full-fledged three-dimensional visual programming language, and we have built two implementations of a Cube environment.

It is important to stress that we use the third dimension *not to enhance the beauty of a program*, but rather to *convey semantic meaning*, to *alleviate the screen space problem*, and to *facilitate new interaction environments*.

One benefit of a 3D visual language is that adding an extra dimension provides for a language which is syntactically richer in the sense of Raeder. One can use different spatial dimensions to express different kinds of information. We adopt this idea in Cube: We use horizontal extent

to express the conjunction of logic formulas and the product of types, and vertical extent to express the disjunction of formulas and the sum of types.

The language is divided into two fragments: a fragment of predicate definitions and logic formulas, and a fragment of type definitions and type expressions. In the logic fragment, terms and atomic formulas are represented by cubes. Cubes that are located in the same xy plane represent the *conjunction* of atomic formulas. So, conjunctions utilize 2 of the 3 dimensions. The arguments of predicate applications (i.e. atomic formulas) are connected by pipes, in other words, they form a data flow diagram. It should be noted that data flow diagrams are inherently at least two-dimensional. By using a three-dimensional framework, we avoid the problem of crossing lines — we can always route a pipe through 3-space so that it does not intersect any other pipe or cube.

The body of a predicate definition is normalized to be in disjunctive normal form; that is, it is a disjunction of conjunctions. We use the z dimension to indicate *disjunctions*. "Stacking" a number of conjunctions, each of which extends in the xy dimension, on top of each other, i.e. in the z dimension, indicates that these conjunctions shall be disjoint.

So, we use the third dimension to group various two-dimensional diagrams together. Alternatively, one could display each diagram in its own 2D window; however, this would require the user to make a conscious effort to mentally integrate various windows into a single formula. The 3D representation, on the other side, shifts this effort to the user's cognitive system.

The very same concept — using a third dimension to encode more information — has been successfully used in program visualization and algorithm animation [6, 14, 80] and scientific visualization at large.

The 3D representation also helps to *alleviate the screen space problem*. In a two-dimensional framework, we would use several windows to display the various clauses of a predicate definition, now, we integrate them all into one three-dimensional object, which we display in a single window. Of course, in the 3D setting, some objects in a scene may be obscured by other objects in front of them; the user can resolve this by rotating the scene.

Another potential reason for visual languages to adopt a three-dimensional notation is that such a notation naturally complements a Virtual Reality (VR) environment. There are two reasons why this might be desirable: one might want to use the VR environment as a pro-

gramming environment, or one might want to use a 3D visual language inside an existing VR environment to program VR simulations.

VR environments have potential as comfortable programming environments due to the very immediate mode of interaction which they allow. Instead of manipulating graphical objects with a mouse, a programmer can literally reach out, grab an object, and move it around. VR environments also promise to alleviate the screen space problem even further. Standard solutions in the two-dimensional setting are zooming, panning, and elision techniques; however, requiring the user to adjust the amount of detail shown through sliders or other interactors adds another layer of complexity to the programming environment. In a VR environment, on the other hand, the user has a much larger "virtual space" available. He can focus onto a different part of the picture simply by turning his head, or by rotating an object in front of him through direct manipulation.

The second reason why one might want to program in a VR environment is in order to develop VR software. This was the motivation for the work on Lingua Graphica [81]. Lingua Graphica was developed at Lockheed's Artificial Intelligence Center. It is part of a larger virtual environment system called Seraphim [27], directed at developing and delivering effective training applications. The key idea is that developing virtual reality software within a virtual reality programming environment should provide for very short edit-compile-debug cycles.

1.5 Semantic Strengths of Cube

The benefits of static type systems are widely recognized. They help detect programming errors statically, which otherwise could be found only through exhaustive testing of every part of the program, and thus they contribute to faster program development, fewer debugging cycles, and more reliable code.

In the best of all possible worlds, the semantics of a language guarantees that a well-typed program will never "go wrong" at run-time, that is, it will never fail due to a type error. Such languages are called *type-safe*. The notion of type-safety does not cover errors like "division-by-zero" (which could be regarded as a type error!) or non-termination of a program.

Most procedural languages like Fortran or Pascal are not completely type-safe; they do not prevent errors such as trying to access non-existing array elements. Most modern functional

languages, on the other hand, have a static polymorphic type system which guarantees type-safety. These languages typically use the Hindley-Milner inference algorithm [17] to determine whether a program is well-typed. The Hindley-Milner algorithm is a type-reconstruction algorithm, which means that the user does not have to specify the types of variables (or other expressions) in a program; instead, the Hindley-Milner algorithm *infers* them through a combination of unification and natural deduction.

Cube has a static polymorphic type system, it is type-safe, and it uses the Hindley-Milner algorithm to reconstruct types. Ill-typed programs are rejected, and the reconstructed types of well-typed programs are visualized to provide feedback to the user (in this respect, Cube is more user-friently than most textual languages that use Hindley-Milner; typically, they do not provide such feedback).

To our best knowledge, we were the first to incorporate the Hindley-Milner algorithm into a visual language [56], and to make strong guarantees about type-safety in a visual setting.

Horn logic was first proposed as a programming language by Robert Kowalski, whose seminal book "Logic for Problem Solving" [40] laid the foundations for an entire branch of programming language research, namely logic programming. The basic idea of logic programming is that predicate logic is a powerful formalism for describing problems in a declarative way. Unfortunately, automatic proof methods for unrestricted predicate calculus sentences are exceedingly expensive. But if we restrict ourselves to sentences of a certain form, namely conjunctions of Horn clauses (which in turn are disjunctions of literals with at most one positive literal), we can use proof methods which are efficient enough to form the basis of a programming language. One particular such proof method, SLD resolution, is used by Prolog [13], the most popular logic programming language to date.

Prolog is a first-order language: predicates are the "agents of computation", but they are not first-class objects. They cannot be passed around as arguments to other predicates and then be applied by the other predicate (Prolog has a metapredicate "call", which solves this problem, but relies on some very unclean features in the process). In other words, Prolog misses out on the tremendous abstraction features provided by higher-orderness, and on the payoffs in code reduction and code reuse. Several newer logic programming languages have been developed to solve this problem. Cube is one of them, it treats predicates as first-class values, and allows the user to define higher-order predicates. While our approach is quite simplistic, it is powerful

enough to allow us to formulate all the higher-order idioms from the functional programming world in Cube; moreover, it is very efficient.

1.6 Contributions

The previous sections have described the strengths and the weaknesses of visual languages, they have given arguments in favor of a 3D notation, and they have described the contributions of Cube in a narrative form. Some of these contributions relate to its novel, three-dimensional syntax, others to its powerful semantics. Features from both sides contribute to overcoming the aforementioned weaknesses of existing visual languages.

At this point, we may summarize the contributions of this thesis concisely as:

- Cube is the first three-dimensional programming language. It uses the third dimension in a meaningful way, namely to encode semantic information. The use of a 3D syntax also opens up the door for novel, virtual-reality-based programming environments.
- We address the screen space problem by placing programs in a three-dimensional space as opposed to on a two-dimensional plane.
- Cube uses a static type inference system, and guarantees that well-typed programs will not "go wrong" at run-time.
- It is based on a very powerful semantic model, namely a higher-order form of Horn logic. This model provides higher-order operations, unification, multi-directionality, static scoping, nested definitions, and implicit parallelism on the language level.
- We apply the data flow metaphor to logic programming. Data flow is one of the most appealing visual metaphors; however, none of the visual logic languages we are aware of uses it.
- The implementations translate visual programs into a textual notation, and then operate on this textual notation, thereby debunking the myth that visual languages are intrinsically less efficient than textual ones.

1.7 Disclaimers

This thesis does not cover a number of aspects which will have to be addressed in order to ultimately decide whether or not 3D visual programming is an improvement over more conventional techniques.

Our prototype implementation falls short of the envisioned ideal environment, in that it is not implemented on an actual Virtual Reality platform. Therefore, the interaction environment, especially the editor, is rather tedious to use. Selecting a point in 3D is trivial, if one has a 3D input device, but it is rather cumbersome when using a 2D device such as a mouse (Section 6.2.2 elaborates on the techniques we use for 3D point specification). As a result, the construction of even a simple Cube expression takes a fair amount of time, certainly longer than the construction of the same expression in a 2D visual or in a textual language.

In fact, our prototype system performs all the 3D rendering in software (although it would be easy to utilize 3D graphics hardware). Hence, generation of a high-quality rendering takes around 10 seconds on a Sun SPARCstation 10. When the user moves around in a scene, the system falls back onto a wireframe representation, which can be generated in real time. The wireframe graphics, however, is rather hard to comprehend, and thereby forms another obstacle regarding the system's usability.

These two deficiencies of the current implementation explain why we did not perform any user studies. Ultimately, however, the usefulness of any new programming language and programming environment can only be determined through such empirical studies.

1.8 Thesis Outline

In the next chapter, we review related work on functional and on logic visual languages, on typed and on higher-order visual languages, and on other forays into 3D in the areas of visual programming and of program visualization. We also review work done in applying static type inference to logic programming languages, and work on higher-order logic languages.

Chapter 3 gives an account of the evolution of Cube's syntax, and argues that the choice of a three-dimensional syntax was a natural one. It describes how our semantic model resulted from merging features of Prolog with those of a typed, higher-order functional language such as Lazy ML.

Chapter 4 gives an informal, example-driven introduction to the Cube language. Chapter 5 then formalizes the syntax and the semantics of the language, by relating the visual language to a textual counterpart, and then giving a type system and an operational semantics for this textual language.

Chapter 6 describes the two existing implementations of a Cube environment: the initial feasibility study CUBE-I, which consists mainly of an interpreter, a type inference system, and a renderer, and the prototype implementation CUBE-II, which improves on CUBE-I by offering better performance and by adding interactive features, such as a rudimentary editor.

Chapter 7 finally sums up our findings, and discusses areas of further research.

Chapter 2

Related Work

This chapter reviews previous work which influenced the design of Cube, as well as related work that shares some of Cube's more unusual features. Some features of Cube and its predecessor, ESTL, have in turn influenced the design of other visual languages.

2.1 Show and Tell

The design of Cube was quite heavily influenced by Show and Tell [36, 37, 49], a visual language based on the data flow paradigm. Constants, variables, and operations are shown as *boxes*. Data flows from boxes to other boxes through *pipes*, which are depicted as arrows. A picture composed of boxes and pipes is called a *puzzle*. Show and Tell tries to *complete* this puzzle by performing every possible data flow. If data flows into a box already containing a different value, the box becomes *inconsistent*. Inconsistency can be limited to a single box, or it can "flow out" of this box and turn its spatial environment inconsistent as well. Inconsistent areas are shaded grey and are considered to be removed from the diagram. If a pipe leads through an inconsistent area, no data can pass through it. This novel notion of consistency can be utilized in many ways, in particular, it fulfills the same purpose as a conditional or selection function in traditional textual languages. Cube generalizes the notions of completion and consistency to unification and satisfiability.

Figure 2.1 shows a Show and Tell program for computing the factorial of a number.

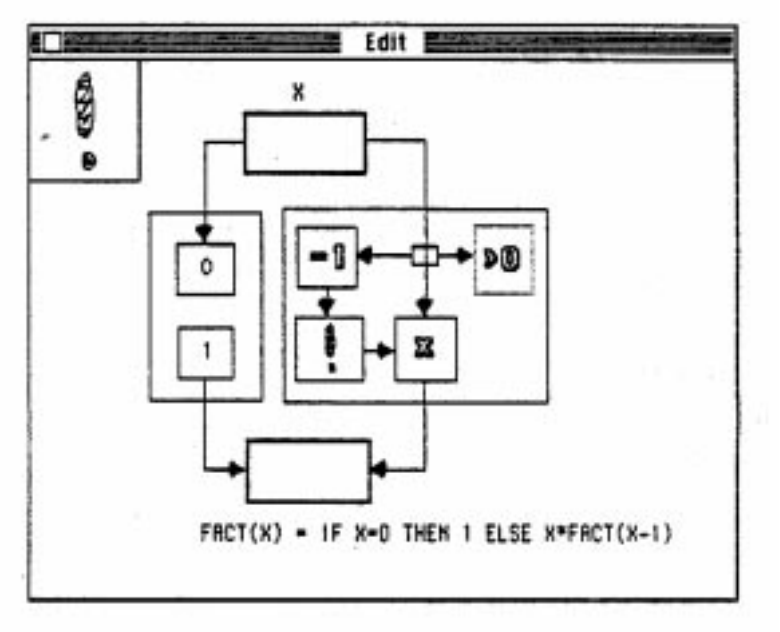


Figure 2.1: Show and Tell Definition of Factorial

2.2 Typed Visual Languages

There are a few visual languages that have an explicit notion of types. They can be divided into two categories: those whose type system is modeled after traditional procedural languages, and those whose type system is based on the Hindley-Milner approach [17, 51].

The first category contains two languages: Fabrik [32, 47] and DataVis [29, 30]. Both of these languages are based on the data flow model. Fabrik is a general-purpose programming language, while DataVis is targeted towards scientific visualization.

Both of them emphasize *concreteness* in their type systems. They feature a rich set of predefined types, such as enumerations, records, arrays, points, bitmaps, etc., but do not allow the user to define new types. In DataVis, types are associated with colors. Two boxes or ports can be connected only if their types match, and in this case, the link appears in the color associated with its type. This approach is limited in two ways: first, the mapping of types to colors assumes a fixed and fairly small set of types, and second, the approach does not deal well with polymorphic operations (although DataVis uses the color white to denote unknown types).

Another severe drawback of the type inference system employed by Fabrik and by DataVis is that it does not guarantee type safety: programs which are judged to be well-typed can still produce run-time errors that are essentially type related. For instance, they might try to reference a non-existing array element.

The second category contains four languages: Enhanced Show-and-Tell, Cube, VisaVis and an extension of Forms/3.

Enhanced Show-and-Tell [56], or ESTL for short, improves on Show and Tell by adding a static polymorphic type system and higher-order functions. It was the first visual language to use Hindley-Milner type inference. Cube [57, 58] in turn is a successor of Show and Tell and ESTL, it transfers some key ideas from these two languages to visual logic programming.

VisaVis [65] is a visual language with a data flow based syntax, whose semantics is based on FFP, the higher-order version of FP [3]. Its type inference system is based on Wand's type inference algorithm, which in turn is a variation of the Hindley-Milner algorithm. Recently, Poswig and Moraga modified the inference system to perform incremental type inference and to support overloaded functions [66].

Forms/3 [7], finally, is a form-based visual language. Burnett's type inference system for Forms/3 [8] is based on set constraints; the inference algorithm is again a variation of the Hindley-Milner algorithm.

All the languages in this second category share one important property: they guarantee type safety, that is, a well-typed program will not fail at run time due to a type error.

2.3 Higher-Order Visual Languages

Apart from adding a type system to Show and Tell, ESTL also introduces higher-order functions. The naming part of a function definition may contain a hole, called a *function slot*, and the body of the definition can contain operators referring to this slot. Other expressions can use the higher-order function, they will then fill the slot with some type-compatible function.

The concept of using function slots to provide arguments to higher-order functions is adopted by DataVis and by VPL [44], a data flow language used for interactive image processing.

VisaVis uses a different metaphor, it refers to function slots as to "keyholes" and to lower-order functions as to "keys"; however, the basic idea of embedding one function's icon into that

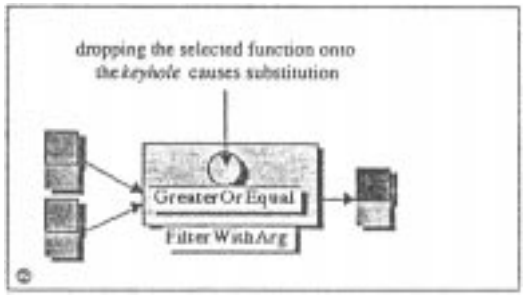


Figure 2.2: Higher-Order Function in VisaVis

of another function remains unchanged. Figure 2.2 shows a VisaVis higher-order function with a first-order function in its "keyhole".

Another visual language which allows for higher-order functions is Holt's *viz* [31], a visual notation for the λ calculus. *viz* treats first- and higher-order functions uniformly; values (functions and non-functions) are denoted by boxes, and application is denoted by stacking the argument (or arguments) on top of the functor.

Cube combines ESTL's function slot concept with *viz*'s uniform treatment of first- and higher-order arguments. There is no distinction between function (or rather, predicate) slots and argument ports; argument values can either be filled directly into a port ("icon inside an icon") or be supplied via a pipe (the classical data flow approach).

2.4 Logic-Based Visual Languages

There are a wide variety of visual logic programming languages. Strangely enough, however, none of them use the data flow metaphor.

A fair number of these languages use a syntax which is based on AND/OR trees. In a sense, this choice of notation is very straightforward. Kowalski himself proposed it in his seminal "Logic for Problem Solving" [40], which laid the foundation for logic programming. On the other hand, while AND/OR trees are well suited to visualize the SLD resolution process, i.e. the unfolding of a logic program as it runs, they are not particularly well-suited for giving an intuition of the meaning of a *static* program.

Languages in this group are the Transparent Prolog Machine [20], Senay and Lazzeri's system [74], and VPP [63]. Figure 2.3 shows the visualization of a clause depicted by Senay

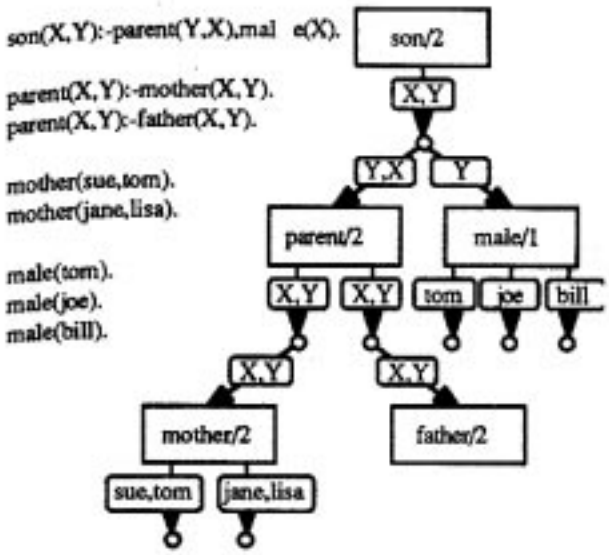


Figure 2.3: Senay and Lazzeri's System

and Lazzeri's system. It illustrates another problem with the AND/OR tree approach: while it visualizes the relationship between the head of a clause and its subgoals, it does not visualize other important aspects, such as multiple occurrences of the same shared variable. Incidentally, this figure contains a mistake (the subgoal "male(X)" is visualized as "male(Y)"), which the authors most likely would have discovered, had they used a more visual way to denote variable sharing, such as a data flow diagram.

VLP [41] introduces a number of interesting concepts: Clauses and literals are depicted as boxes, horizontal arrangement of boxes denotes conjunction, while vertical arrangement denotes disjunction. Spatial enclosure is used for "procedural abstraction", i.e. predicate definition. All these concepts are used in Cube as well; we derived them, however, without being aware of Ladret and Rueher's work.

On the other hand, VLP uses shared patterns to indicate shared variables. This approach makes it quite hard to discover all occurrences of the same variable; a data flow notation would alleviate the problem significantly. However, data flow diagrams are an inherently two-dimensional notation, and Ladret and Rueher's decision to use one dimension to indicate disjunction leaves them with just one more dimension for conjunctions, i.e. clause bodies, where

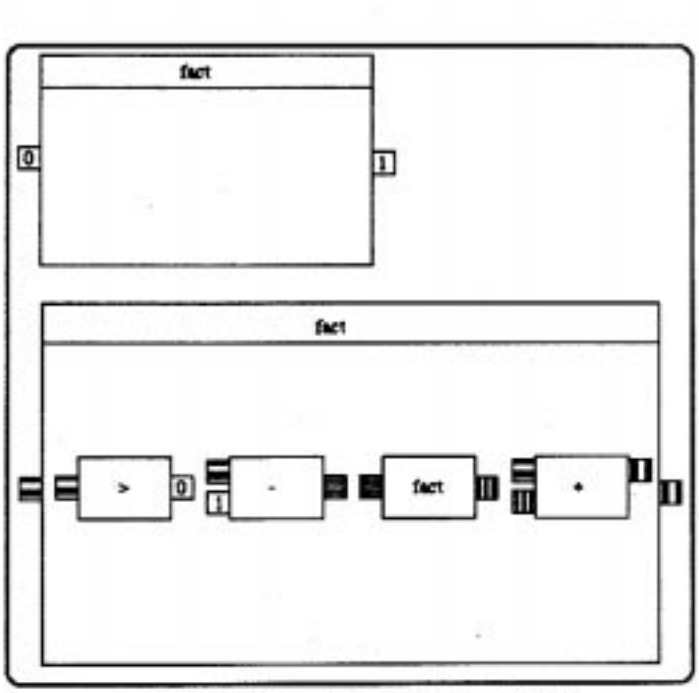


Figure 18. A standard representation of fact/2

Figure 2.4: VLP Definition of Factorial

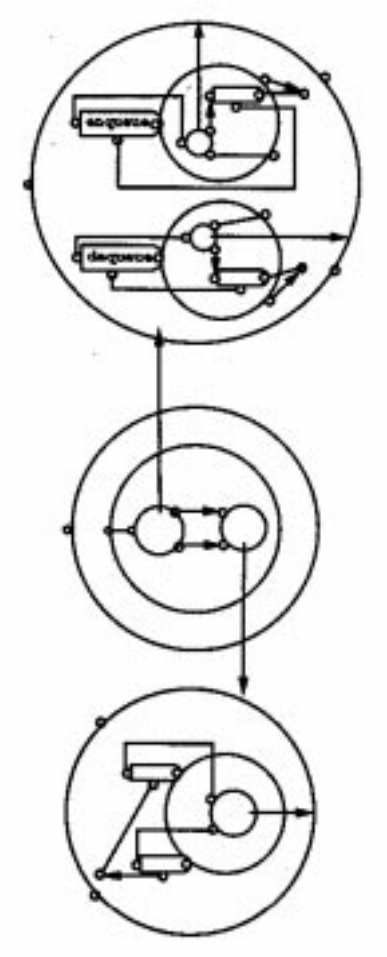


Figure 2.5: Pictorial Janus

most of the occurrences of a shared variable are typically located. Cube overcomes this problem by moving into 3D. Figure 2.4 shows a VLP definition of the factorial predicate.

Pictorial Janus [35] is a visual notation for Janus [72], a concurrent constraint logic language. It offers a powerful and elegant visual metaphor for the underlying resolution process: diagram rewriting. Predicate definitions, clauses, and subgoals are represented as closed contours (such as circles). Whenever one particular clause is selected to replace a goal, its contour is slowly transformed to replace the contour of the original goal. This rule is not only simple, but it also leads to a fluid animation of the resolution process. On the flip-side, a query can mutate considerably during the rewriting process, so that it can be hard for the user to determine

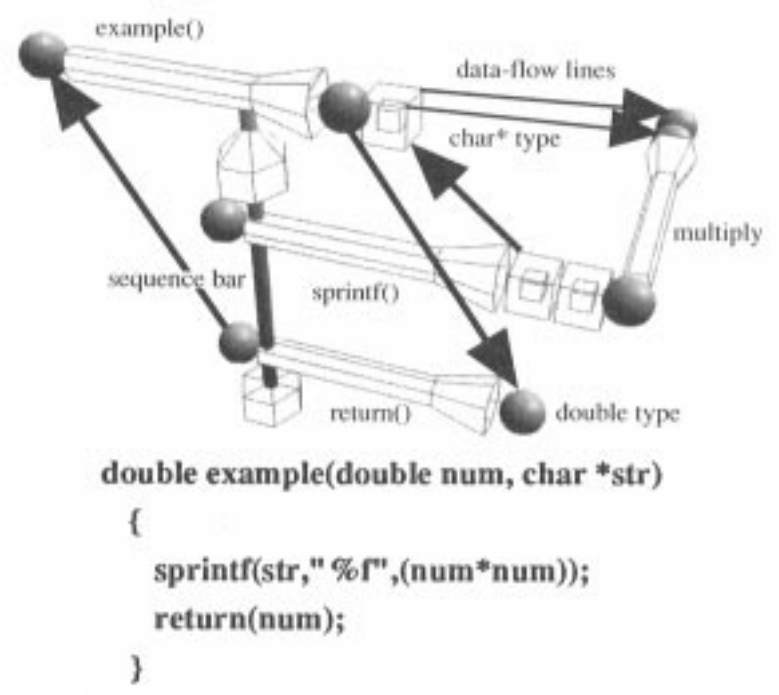


Figure 2.6: Lingua Graphica

the spatial correspondences between the final result and the initial query. Figure 2.5 shows a Pictorial Janus program for implementing a queue.

SPARCL [79] is visual logic language based on sets. While other logic programming languages use uninterpreted functions (i.e. constructors) to combine symbols into larger data structures, SPARCL uses sets to aggregate data. It also allows its user to divide a set into pairwise disjoint subsets. Spratt and Ambler show that they can model conventional tuples and terms with their set notation, and that the approach is thus sufficiently powerful.

2.5 3D in Visual Programming and in Program Visualization

The potential for 3D visual languages has been realized early on by Glinert [24], who argued that they are a natural next step in the evolution of visual programming, and that they might provide better visual metaphors than 2D languages can provide. Cube was the first such language. Recently, it has been joined by two other 3D visual languages, Lingua Graphica and CAEL-3D. Both of them are visual layers put on top of an existing procedural textual language.

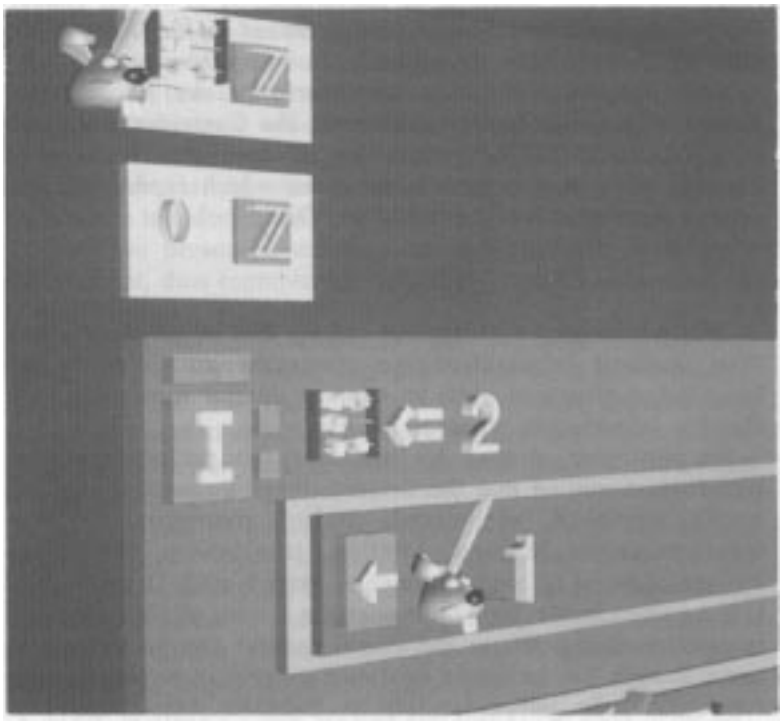


Figure 2.7: CAEL-3D

Lingua Graphica [81] is a visual language which provides a 3D syntax for C++ programs. Created at Lockheed AI Labs, it is intended to allow Virtual Reality operators to inspect and modify VR simulation code without having to leave the virtual environment. Figure 2.6 shows an example of a Lingua Graphica program.

The interactive "Computer Animation Environment Language" CAEL is a textual language, which augments a subset of Pascal with procedures that allow its user to describe arbitrary 3D animations. CAEL-3D [68] is a 3D visual syntax for CAEL. Figure 2.7 shows a CAEL-3D example program.

Campanai, Del Bimbo, and Nesi [9, 18] have been using a 3D query language and a virtual reality setup to access the contents of a database storing 3D scenes. The basic idea here is that the user constructs a virtual scene from existing 3D icons (such as houses, trees, and cars), using a data glove as the input device, and that this scene is then matched against the contents of the database to access all the stored scenes containing a matching arrangement of objects. Figure 2.8 shows a view of their system.

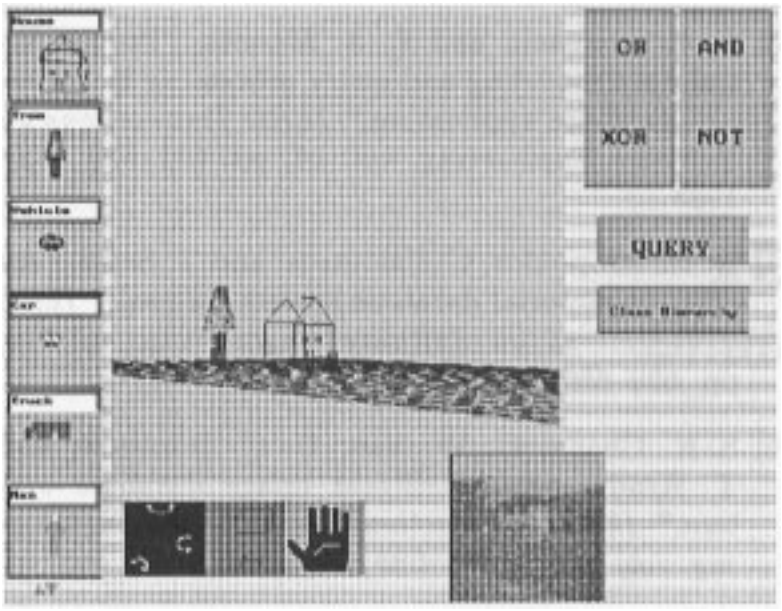


Figure 2.8: Campanai, Del Bimbo, and Nesi's System

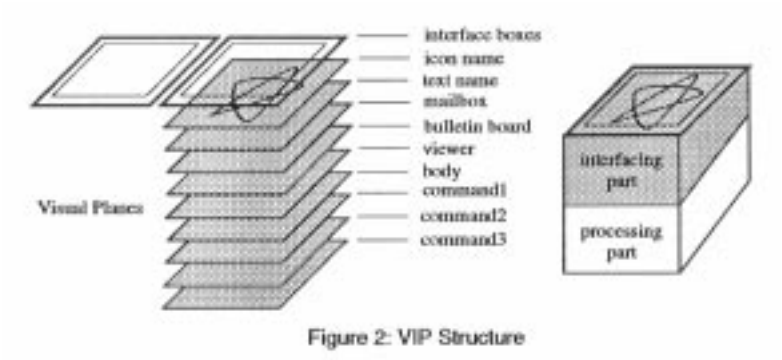


Figure 2.9: Hyperflow's Metaphor of Stacked "Visual Planes"

Kimura's Hyperflow [38] is a 2D visual language with a data flow syntax and an object-oriented semantics. Although the language is two-dimensional, it employs a three-dimensional metaphor: Objects are viewed as stacks of diagrams; and each diagram shows a different aspect of an objects definition. Figure 2.9 illustrates this metaphor. The diagram stack metaphor played a key role in the development of Cube's syntax.

There are various program visualization and algorithm animation systems which use 3D graphics to visualize the behavior of a running program. The earliest such system we are aware of is Lieberman's system for visualizing the execution of Lisp programs [45]. The view shows the code for an expression on the front side of a block. As the expression gets evaluated, each application causes a smaller block with the corresponding code to be displayed in front of the caller's block. When an expression is evaluated, its block is removed.

Pavane [14, 71] supports both 2D and 3D views of concurrent computations. Its formalism is based on a combination of Prolog clauses and Linda's tuple space concept. As a computation unfolds, it adds tuples to the tuple space which characterize its internal state. A visualizer continuously retrieves these tuples and uses an application-specific set of rules to map them into 2D or 3D pictures. The system also supports smooth transitions from one state to the next.

Zeus3D [6] is a 3D extension of the Zeus algorithm animation system. Algorithms in Zeus are annotated with event-generating procedures, these events then cause various views associated with the algorithm to be updated. Polka-3D [80] is a 3D extension of the Polka algorithm animation system. Its basic philosophy is quite similar to Zeus3D.

Finally, Plum [69] is a package to visualize abstract data in 3D. Plum has been used to visualize static and dynamic properties of programs, such as their call graphs.

2.6 Higher-Order Logic Languages

Since the early 1970's, logic, and in particular Horn logic, has been proposed as a programming language [40], which eventually led to the development of Prolog [13], a textual language based on Horn logic.

There have been several attempts to add higher-order features to the first-order logic of Prolog. The first higher-order logic programming language we are aware of is λProlog by Miller and Nadathur. In its initial form it was based on higher-order definite clauses [50].

λProlog does not distinguish between terms and atomic formulas. A term can be a variable, a constant (constructor or predicate) symbol, an application, or a λ abstraction. So, the term fragment of the language corresponds to the lambda calculus.

In the initial version of λProlog, the formula fragment corresponded precisely to Horn formulas, i.e. Prolog programs, except that Prolog's terms got replaced by this richer notion of terms, and that terms could appear in place of atomic formulas. A later version of the language [53], based on higher-order hereditary Harrop formulas, uses a richer syntax for the formula fragment of the language as well.

The proof mechanism employed by λProlog is considerably more powerful than the one used by Cube. For example, given an appropriate definition of map, a well-known higher-order predicate, and the query "map F $[1,1]$ $[(\mathsf{g}\ 1\ 1),(\mathsf{g}\ 1\ 2)]$", λProlog will infer a solution for the variable F, namely $\lambda x\,.\,\mathsf{g}\ 1\ x$. Cube, on the other side, would simply suspend. λProlog achieves this extra degree of power by using a higher-order form of unification: it is able to compute a unifier for the equation "$(F\ 1) = (\mathsf{g}\ 1\ 1)$". However, higher-order unification is in general only undecidable [26], the search for a non-existing unifier may lead to divergence.

λProlog uses a curried notation for predicate applications, and features an ML-like type inference system. Predicates are viewed as functions mapping to truth values, for example, map has the type $(A \to B) \to (\mathit{list}A) \to (\mathit{list}B) \to o$ (where A, B are type variables and o is the type of propositions). Cube adopted these two features from λProlog.

A prototype interpreter for λProlog has been implemented, and a more efficient, abstract-machine based implementation is under way [54].

HiLog [12] is a logic programming language with a higher-order syntax, but a first-order semantics. There is no distinction between terms and atomic formulas. Two terms denoting predicates are considered to be equal if their intensions are equal (e.g. if they are denoted by the same symbol); no attempt is made to decide if their extensions (i.e. the relations they describe) are equal (the latter is in general undecidable [26]).

A term which is an application may in turn be applied to another term; in other words, HiLog allows for a curried style of programming.

HiLog appears to be more powerful than Cube, it features universal and existential quantification as well as negation. Chen, Warren, and Kifer give a model theory as well as a proof theory for HiLog; however, they do not mention any implementation of the language.

Andrews' logic G [1] is the third attempt we know of to incorporate higher-order features into logic programming. G, just like HiLog, has a higher-order syntax and a first-order semantics. Predicate names are considered to be terms; unification determines intensional identity, not extensional equivalence between two terms. Other additions to the syntax of terms are tuples and set abstraction terms. Predicates are regarded as sets of term tuples, predicate application is thus viewed as set membership test.

Andrews gives both a model theory and a proof theory for the language. In addition, he provides an operational semantics, which is similar to the standard semantics of prolog given by Lloyd [46].

The proof mechanism of G is more powerful than the one of Cube. Given an appropriate definition of the higher-order predicate Map, G is able to satisfy the query "$\exists x \,.\, \mathsf{Map}(x, [\mathsf{a}, \mathsf{b}], [\mathsf{c}, \mathsf{db}])$", and derives a substitution of "$\{y \mid y = \langle a, c\rangle \vee \langle b, d\rangle\}$" for x. Given the same query, Cube would simply suspend. On the other side, G neither allows for terms with variable functor, nor for curried applications, both of which are permissible in Cube. As far as we know, no implementation of G exists so far.

2.7 Type Inference Systems

Milner developed a static type inference system for the polymorphic lambda calculus [17, 51], known as the Hindley-Milner type system, which has since then been widely used for functional languages. Mycroft and O'Keefe adopted it for Prolog [52], and recently Lakshman and Reddy [43] gave a semantic foundation to this adaptation. λProlog also uses Milner's type system.

Cube's type system is based on Milner's system as well. Thus it resembles the one used by λProlog. It differs from the one proposed by Lakshman and Reddy by viewing predicates as functions mapping to truth-values, and thus assigning a function type to them. This view is convenient in the presence of curried predicate applications.

Chapter 3

Motivation

The development of Cube was driven by two basic goals: To find a better visual syntax for logic programming, and to make an existing logic programming language – Prolog – cleaner, safer, and more expressive.

This chapter explains some of the motivations behind this thesis. We account how Cube's syntax evolved from the two-dimensional notation of Show and Tell into its current three-dimensional form, and we explain how Cube's semantics resulted from combining Prolog with features of higher-order functional languages.

3.1 Evolution of Cube's Syntax

During our work on ESTL [56], we realized that Show and Tell embodies many ideas of logic programming in a weakened form. One of its key concepts is the notion of consistency: during an evaluation of a data flow diagram (a boxgraph in Show and Tell terminology), parts of the diagram may become inconsistent, and are then considered to be removed from the computation. There are two possible causes for inconsistency: two conflicting values flow into the same box, or a relational predicate fails. The former resembles a failed unification, the latter a failed predicate application in logic programming.

Another key concept is the notion of completion: a data flow diagram may contain a number of empty boxes, which may receive a value during evaluation, but do not have to. In the simple case, these values are immutable: once a box receives a value, it will not be changed (Show and

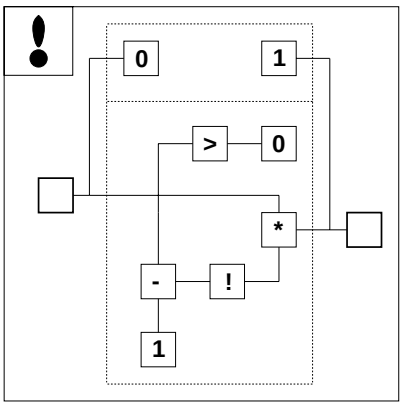


Figure 3.1: Adding Undirected Links and Disjunction to Show and Tell

Tell departs from this rule when showing the evaluation of certain "iterators"). This notion of completion resembles the concept of unification in logic programming.

Other features of Show and Tell, however, are more restrictive than those found in Prolog. The links of a data flow diagram are directed, data can flow only in the direction of the links. Hence, this model of data propagation is less powerful than unification, where information can flow in either direction.

Second, Show and Tell has no notion of disjunction. All the parts of a diagram contribute to the same solution. If one uses the concept of inconsistency to perform a case analysis, he must be sure that the various cases exclude each other, otherwise the results of two matching cases might collide when they are merged.

We initially tried to transform Show and Tell into a two-dimensional visual logic language by generalizing data flow links to be undirected, with the semantics that two boxes or ports that are connected by a link are unified, and by introducing a notation for disjunction into the language. Figure 3.1 shows the definition of factorial in this notation.

However, we found that the diagrams were becoming rather confusing. One solution to this problem seemed to be to have a separate data flow diagram for each clause. Figure 3.2 shows the definition of factorial, represented by two diagrams.

In such a framework, a predicate definition would consist of an arbitrary number of "box-graphs". One could put them next to each other, or one could display only one at a time, and provide controls to switch between them. The latter idea gives rise to a stack metaphor: The

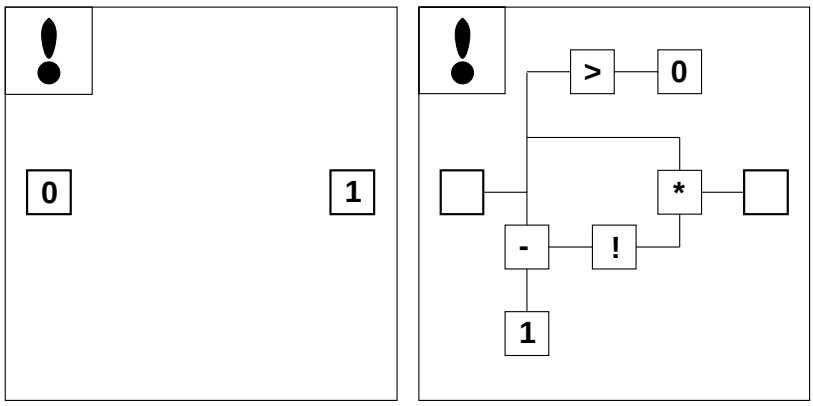


Figure 3.2: Using One Diagram Per Clause

different boxgraphs of a predicate definition are stacked on top of each other, the interactive controls allow the user to browse through the stack. From there, the move to a three-dimensional representation is only a small step.

Kimura's Hyperflow [38], a two-dimensional data flow language with provisions for object-oriented programming, appears to have been influenced by Cube. It uses the very same metaphor of stacked visual planes, where each plane is a diagram which describes a different aspect of the object.

3.2 Evolution of Cube's Semantics

One of the most interesting features of modern functional languages is their treatment of functions as first-class objects. This feature allows for the definition of higher-order functions, functions that take other functions as arguments.

Suppose we want to define a function map which behaves as follows:

$$\mathsf{map}\ f\ [e_1, \cdots, e_n] \ = \ [f\ e_1, \cdots, f\ e_n]$$

i.e., map applied to a function f and a list of elements $e_1, \cdots, e_n$ returns the list resulting from applying f to each element in the list.

In a functional language such as Lazy ML, we could define map in a recursive fashion as follows:

$$\mathsf{map}\ f\ [\,]\ =\ [\,]$$
$$\mathsf{map}\ f\ (x\,.\,l)\ =\ (f\ x)\,.\,(\mathsf{map}\ f\ l)$$

Prolog, on the other side, is a first-order language, and therefore does not allow for such an elegant definition of the map function. One can, however, use the built-in metapredicate call to achieve a similar effect:

$$\mathsf{map}(F, [\,], [\,]).$$
$$\mathsf{map}(F, (X\,.\,L), (X'\,.\,L')) \ :\Leftrightarrow P\ =..\,[F, X, X'], \mathsf{call}(P), \mathsf{map}(F, L, L').$$

$=..$ (pronounced "univ") is a binary infix predicate that relates a structure to a list. In particular, the relation

$$f(t_1, \cdots, t_n)\ =..\,[f, t_1, \cdots, t_n\,]$$

holds. call takes a term, interprets it as a goal (i.e. a predicate application), and tries to prove it.

The base case of this predicate definition is analogous to the base case of the functional definition: Mapping a predicate F over the empty list yields the empty list. The recursive case, however, is more convoluted: The head of the clause takes a predicate F and a list, which it decomposes into a head X and a tail L, and returns a result list consisting of head X' and tail L'. The subgoal $P\ =..\,[F, X, X']$ unifies P with the term $F(X, X')$, $\mathsf{call}(P)$ then interprets P as a goal and tries to prove it. In other words, $P\ =..\,[F, X, X'], \mathsf{call}(P)$ holds if $F(X, X')$ holds. $\mathsf{map}(F, L, L')$ finally applies map recursively to F and L, yielding L'.

There are two problems with this approach: First, the list $[F, X, X']$ is usually heterogeneous, which makes it impossible to apply standard type checking algorithms such as [43]. And second, P, which is formally a term, really denotes a predicate application, and F, which is formally a term, really denotes a predicate symbol.

There is an obvious solution to those two problems: Lift the distinction between terms and predicates, and allow variables to occur as functors of terms. Given these two modifications, we can reformulate our definition of map as follows:

$$\mathsf{map}(F, [\,], [\,]).$$
$$\mathsf{map}(F, (X\,.\,L), (X'\,.\,L')) \ :\Leftrightarrow F(X, X'), \mathsf{map}(F, L, L').$$

This definition looks much more similar to the functional definition of map. The main remaining difference is that in the logic framework, the body of a clause consists of a "flat" conjunction of subgoals, whereas in the functional framework, the body of a function definition may consist of nested function applications. The flat notation makes clauses harder to read, and it introduces extra variables used to connect the producer of a value to its consumer (in this case, X' and L'). The visual notation used by Cube, however, alleviates these problems to a large degree.

There is, however, a more serious problem: If we treat predicates and predicate symbols as terms, then we can also unify them. The expression

$$\mathsf{fact} \;=\; \mathsf{fact}'$$

should be true if fact and fact′ are equal. We could interpret "being equal" as meaning "describing the same relation". Unfortunately, this question is in general only semi-decidable [26]. Therefore, we instead interpret "being equal" as "being defined in exactly the same way". This question can be answered efficiently (through first-order unification), and it appears as if this restrictive notion of equality is powerful enough to bring all the forms of higher-orderness exploited by functional languages to the logic programming world.

This modification of Prolog allows us to use Hindley-Milner type inference even for higher-order predicates such as map. The type of map, in the notation of Lakshman and Reddy [43], is:

$$\mathsf{Pred}(\mathsf{Pred}(\alpha,\beta),\mathsf{List}(\alpha),\mathsf{List}(\beta))$$

Alternatively, we can view predicates as functions that map to propositional values, i.e. truth or falsity. If we denote the type of propositions by Prop, the type of map is:

$$(\alpha \times \beta \to \mathsf{Prop}) \times \mathsf{List}(\alpha) \times \mathsf{List}(\beta) \to \mathsf{Prop}$$

In many functional languages, an $n$ary function is expressed as a unary function mapping to an $(n \Leftrightarrow 1)$ary function. For example, the map-function

$$map \;=\; \lambda(f,\mathrm{a})\,.\,\mathbf{if}\ \mathrm{a} = [\,]\ \mathbf{then}\ [\,]\ \mathbf{else}\ (f\ (hd\ \mathrm{a}))\,.\,(map\ (f,(tl\ \mathrm{a})))$$

is instead written as

$$map' \;=\; \lambda f\,.\,\lambda a\,.\,\textbf{if}\ a = [\,]\ \textbf{then}\ [\,]\ \textbf{else}\ (f\ (hd\ a))\,.\,((map'\ f)(tl\ a))$$

This technique, known as *currying*, can be quite useful. Suppose we want to define a function *sqrlist* which computes the square of all elements of a list. In the absence of currying, we would define it as

$$sqrlist\ l \;=\; map\ (sqr, l)$$

Using currying, however, the definition simplifies to

$$sqrlist' \;=\; map'\ sqr$$

The type of map is $((\alpha \to \beta) \times \mathsf{List}\ \alpha) \to \mathsf{List}\ \beta$, the type of map' is $(\alpha \to \beta) \to (\mathsf{List}\ \alpha \to \mathsf{List}\ \beta)$. $\to$ is a right-associative operator, so we can omit the last pair of parentheses. Similarly, application (denoted by juxtaposition of functor and argument) is a left-associative operation, and again we can frequently omit some of the parentheses.

We adopt these ideas to our derivate of Prolog. We use juxtaposition to denote application, that is, we write $f\ a_1\ \cdots\ a_n$ instead of $f(a_1, \cdots, a_n)$. The definition of map therefore changes to

$$\mathsf{map}\ F\ [\,]\ [\,].$$
$$\mathsf{map}\ F\ (X\,.\,L)\ (X'\,.\,L')\ :\Leftrightarrow F\ X\ X', \mathsf{map}\ F\ L\ L'.$$

and its type to $(\alpha \to \beta \to \mathsf{Prop}) \to \mathsf{List}\ \alpha \to \mathsf{List}\ \beta \to \mathsf{Prop}$.

We also introduce an abstraction construct λ. Now we might try to define the map predicate as follows:

$$map \;=\; \lambda f\,.\,\lambda a\,.\,\lambda b\,.\,(a = [\,] \wedge b = [\,]) \vee (a = x\,.\,l \wedge b = x'\,.\,l' \wedge f\ x\ x' \wedge map\ f\ l\ l')$$

While the use of the λ-abstraction apparently made our definition more complex, it also brought a quite fundamental change of view. Beforehand, we viewed map as a constant symbol denoting

a predicate, now, we view it as a variable which gets bound through unification to an anonymous predicate![1]

It turns out, however, that this brings along a new problem: By the definition of unification, there is no unifier for an equation of the form $x = f(\cdots, x, \cdots)$. So, we cannot express *map* as shown above. However, introducing a fixed-point operator solves this problem, as it allows us to transform the recursive definition of *map* into a nonrecursive form:

$$map = \textbf{fix}\ map'\,.\,\lambda f\,.\,\lambda a\,.\,\lambda b\,.\,(a = [\,] \wedge b = [\,]) \vee (a = x\,.\,l \wedge b = x'\,.\,l' \wedge f\ x\ x' \wedge map'\ f\ l\ l')$$

To make the formulation of predicate definitions — recursive and non-recursive ones — easier, we introduce a piece of syntactic sugar, namely the **letrec** binding construct. Now we can write *map* as

$$\textbf{letrec}\ map = \lambda f\,.\,\lambda a\,.\,\lambda b\,.\,(a = [\,] \wedge b = [\,]) \vee (a = x\,.\,l \wedge b = x'\,.\,l' \wedge f\ x\ x' \wedge map\ f\ l\ l')\ \textbf{in}\ \cdots$$

The ellipsis denotes an expression (such as a goal) to which the definition of map is visible. For example, a query which computes the square of all the elements of a list, and which uses the currying technique described above, could be written as:

$$\begin{array}{l}\textbf{letrec}\ map = \lambda f\,.\,\lambda a\,.\,\lambda b\,.\,(a = [\,] \wedge b = [\,]) \vee (a = x\,.\,l \wedge b = x'\,.\,l' \wedge f\ x\ x' \wedge map\ f\ l\ l')\ \textbf{in}\\ \textbf{letrec}\ sqr = \lambda u\,.\,\lambda v\,.\,times\ u\ u\ v\ \textbf{in}\\ \textbf{letrec}\ sqrlist = map\ sqr\ \textbf{in}\\ sqrlist\ [1,2,3]\ z\end{array}$$

Evaluating this query would yield a solution $z = [1, 4, 9]$.

The introduction of **letrec** adds another very powerful feature to our language: the concept of nested scope. An example may demonstrate its usefulness. Consider a predicate to reverse a list. A naive definition (in "classic" Prolog) would be:

$$\begin{array}{l}\mathsf{reverse}([\,],[\,]).\\ \mathsf{reverse}(X\,.\,L,\,Y)\ :\Leftrightarrow \mathsf{reverse}(L, L'), \mathsf{append}(L', [X],\, Y).\end{array}$$

[1] The fact that *map* is now shown in the same font as other variables shall indicate this change of view. Also, starting from this example, I shall use lower-case names for all the variables instead of the upper-case names common in Prolog.

Unfortunately, this definition is rather inefficient; reversing a list with n elements takes $O(n^2)$ time. The following definition works in $O(n)$ time:

$$\mathsf{reverse}(X, Y) \ :\Leftrightarrow \mathsf{rev}(X, Y, [\,]).$$
$$\mathsf{rev}([\,], L, L).$$
$$\mathsf{rev}(X \,.\, L, Y, Z) \ :\Leftrightarrow \mathsf{rev}(L, Y, X \,.\, Z).$$

In this example, rev is an auxiliary predicate, which is only supposed to be called by reverse. However, in Prolog it is visible to all other predicates. In our language, however, we can write the reverse predicate as follows:

$$\begin{array}{l}\textbf{letrec}\ \mathit{reverse} \ = \ \lambda x \,.\, \lambda y. \\ \qquad \textbf{letrec}\ \mathit{rev} \ = \ \lambda a \,.\, \lambda b \,.\, \lambda c \,. \\ \qquad\qquad (a = [\,] \wedge b = c) \vee \\ \qquad\qquad (a = d \,.\, e \wedge \mathit{rev}\ e\ b\ (d \,.\, c)) \\ \qquad \textbf{in}\ \mathit{rev}\ x\ y\ [\,] \\ \textbf{in}\ \cdots\end{array}$$

Now, the definition of *rev* is visible only to the definition of *reverse*, but invisible to the clients of *reverse* (denoted by the ellipsis).

In Prolog, the variables used in a clause are implicitly universally quantified, i.e.

$$\mathsf{reverse}(X \,.\, L, Y) \ :\Leftrightarrow \mathsf{reverse}(L, L'), \mathsf{append}(L', [X], Y).$$

stands for

$$\forall X, Y, L, L'.\mathsf{reverse}(X \,.\, L, Y) \ :\Leftrightarrow \mathsf{reverse}(L, L'), \mathsf{append}(L', [X], Y).$$

We could take the same stand in our language, that is, we could say that variables which are not introduced by a λ or a **letrec** are implicitly universally quantified. The scope of such a quantification, however, would be the entire program, as we have replaced the set of clauses that makes up a Prolog program by a single expression, usually a **letrec**. This approach would contradict the information-hiding idea which we wanted to promote through the **letrec**-construct in the first place. Therefore, we abolish the idea of implicit universal quantification, and instead introduce the existential quantifier $\exists$ into our language. All variables have to be explicitly introduced, either by a λ, a **letrec** , or an $\exists$.

It is easy to see that universally quantifying a variable within the scope of a clause is equivalent to existentially quantifying it within the scope of a clause body (provided that the variable does not occur in the head of the clause), i.e. that

$$\forall X, Y, L, L'.\mathsf{reverse}(X \,.\, L, Y) \;:\Leftrightarrow \mathsf{reverse}(L, L'), \mathsf{append}(L', [X], Y).$$

and

$$\forall X, Y, L \,.\, \mathsf{reverse}(X \,.\, L, Y) \;:\Leftrightarrow \exists L' \,.\, \mathsf{reverse}(L, L'), \mathsf{append}(L', [X], Y).$$

are equivalent.

The existential quantifier allows us to precisely specify the scope of a variable, and thus makes programs much easier to read. The new and final definitions of *map* and *reverse* are:

letrec $map \;=\; \lambda f \,.\, \lambda a \,.\, \lambda b \,.$
$\quad (a = [\,] \wedge b = [\,]) \vee$
$\quad (\exists x, x', l, l' \,.\, a = x \,.\, l \wedge b = x' \,.\, l' \wedge f \;\; x \;\; x' \wedge map \;\; f \;\; l \;\; l')$
in $\cdots$

letrec $reverse \;=\; \lambda x \,.\, \lambda y \,.$
$\quad$ **letrec** $rev \;=\; \lambda a \,.\, \lambda b \,.\, \lambda c \,.$
$\quad\quad (a = [\,] \wedge b = c) \vee$
$\quad\quad (\exists d, e \,.\, a = d \,.\, e \wedge rev \;\; e \;\; b \;\; (d \,.\, c))$
$\quad$ **in** $rev \;\; x \;\; y \;\; [\,]$
in $\cdots$

The textual language we have developed here is almost identical to the textual version of Cube introduced in Chapter 5. The only differences stem from the fact that this language uses a positional binding strategy — the first actual parameter gets bound to the first formal parameter, whereas Cube and its textual counterpart use named parameters and employ a binding-by-name strategy.

Chapter 4

Cube by Example

The following chapter presents a number of Cube example programs, intended to give the reader an intuitive understanding of the semantics of the language. Chapter 5 will then provide a formal definition of both syntax and semantics.

All the figures shown in this chapter were generated by CUBE-II, a prototype implementation of a Cube programming environment (see Section 6.2). It should be noted that black-and-white figures cannot do full justice to the visualization provided by the system, which is not only in color, but moreover interactive, that is, it allows the user to move through the program.

4.1 The Dataflow Metaphor

Consider the simple program shown in Figure 4.1. It consists of two transparent cubes (which are green in the original picture) that are connected by a *pipe*. The transparent cubes are termed *holder cubes*, they may contain terms (which are represented by cubes as well), and thus correspond quite closely to variables in a textual language.

The left holder cube contains a term: an opaque cube (which is green in the original picture) with the icon "1" on its top side. This cube is called an *integer cube*, and represents the integer 1. The two holder cubes are connected by a *pipe*, which serves as a "conduit" for values. The metaphor we use here is the dataflow metaphor: A value contained in a holder cube flows to all the other holder cubes connected to it[1]. If a holder cube receiving a value is empty, it

[1] This dataflow takes place when triggered by the user. In CUBE-I and CUBE-II, the user presses an "EVAL" button.

Figure 4.1: Value 1 Flowing Into Empty Holder

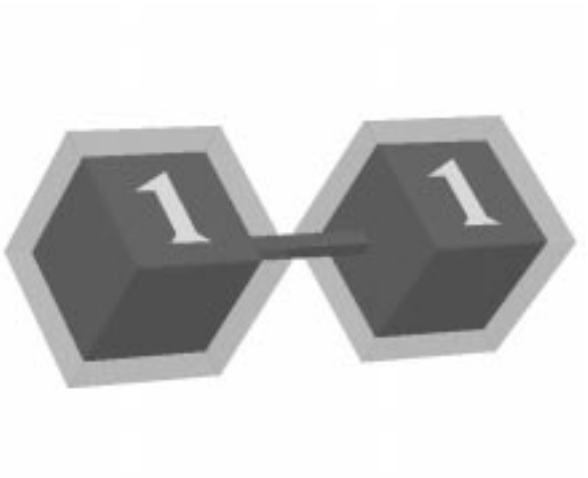


Figure 4.2: Value 1 Has Flown Into Empty Holder

will be filled with this value, if it already contains a value, the two values must be equal (or, more general, *unifiable*), both holder cubes will then contain the same value (namely, the most general unifier of the two values). If this is not possible, the data flow *fails*.

Note that pipes have no particular directionality: data can flow through them in either direction, and as we will see (on page 64), it can indeed flow in both directions at once! It should also be noted that the value contained in a holder cube never gets changed, but only refined.

So, we can extend the analogy we have drawn between Cube and textual languages: Holder cubes correspond to *logic variables*, and connecting two holder cubes by a pipe corresponds to unifying two logic variables. Furthermore, a holder cube containing a term cube (such as an integer cube) corresponds to a logic variable unified with a term.

In the textual framework, a unification is a special case of an atomic formula. A Cube program (i.e. the entire "virtual space" in which Cube expressions are located) corresponds to a query in a textual logic language, that is, a conjunction of all the atomic formulas.

Figure 4.2 shows the Cube program of Figure 4.1 after evaluation. The integer cube 1 flowed from the left to the right holder cube (intuitive interpretation); or the left holder cube got unified with 1 and with the right one, leaving both being instantiated to 1 (logic interpretation).

By contrast, the program shown in Figure 4.3 does not have any solutions: The two holder cubes are connected by a pipe, but one contains the integer cube 1, while the other contains

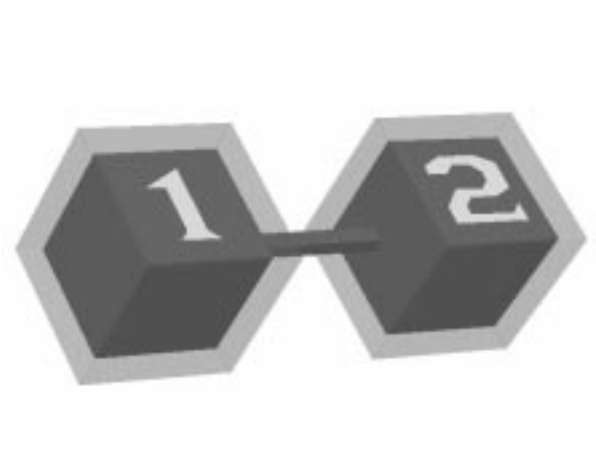


Figure 4.3: Failing Dataflow

the integer cube 2. These two integer cubes are not unifiable, the data flow between the two holder cubes fails, and so does the entire computation.

4.2 A First Glimpse at Types

We have mentioned before that Cube is a statically typed language, and uses a type inference system. In fact, the user can trigger the type inference mechanism at any time. The Cube environment will then determine if the entire program is well-typed. Moreover, if the program is well-typed, it will indicate the type of every empty holder cube by placing a *type cube* inside it. Type cubes are opaque grey cubes with an icon on their top, which identifies the type. There are three predefined based types: Int, the integer type, Float, the floating-point type, and Prop, the type of propositions.

Figure 4.4 shows the type cubes representing these three base types. As we will see later, Cube provides a mechanism that allows users to define other interesting types (such as characters, strings, lists, or trees) themselves.

So, given the program from Figure 4.1, Cube will infer that 1 is an integer, so the left holder cube contains an integer, and therefore the right holder cube must contain an integer as well. It will thus fill the right holder cube with an Int type cube (see Figure 4.5).

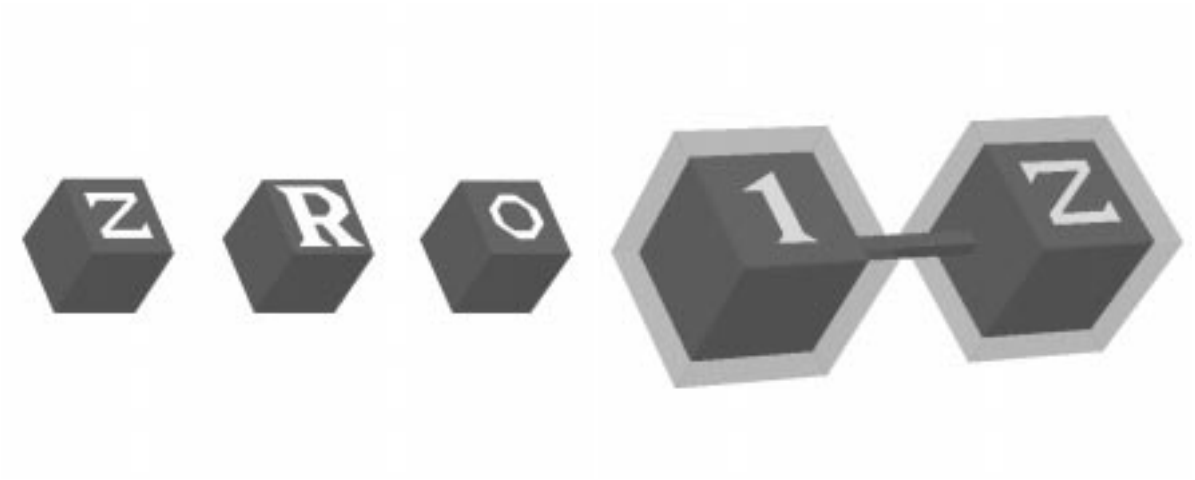


Figure 4.4: Type Cubes of the Base Types

Figure 4.5: Program From Figure 4.1 After Type Inference

4.3 Predicate Applications

The fragment of Cube that we have seen so far hardly qualifies as a programming language — it can move values around, but not perform any computations on them. The device that we are encountering now, however, solves this problem.

Consider the object shown in Figure 4.6, which is called a *predicate cube*. It is represented as an opaque green cube with an icon on its top. The icon identifies the predicate we are referring to (integer addition in this case). The cube also has a number of "holes" in its sides: cubic intrusions with a transparent cover on the outside and an icon on top of it. The "holes" are called *ports* and serve as arguments to the predicate. They may be moved around freely over all 6 sides of the predicate cube; thus, an icon is needed to identify each port. A port is a special case of a holder cube (hence the transparent cover), and as such it can be connected to pipes and can be filled with a value.

It should also be pointed out that the similarity between predicate cubes and integer cubes (both being opaque green cubes with an icon on their top) is not coincidental: both are *term cubes*, they represent a first-class value, i.e. they can both be contained in a holder cube and flow through a pipe. In fact, they are both a form of *reference cube*: a cube that refers to a definition cube visible to it. In the case of integer cubes, we just imagine that a definition of all the integer values is visible. In the case of predicate cubes, however, these definitions are

Figure 4.6: Addition Predicate Cube

Figure 4.7: Type of Addition Predicate

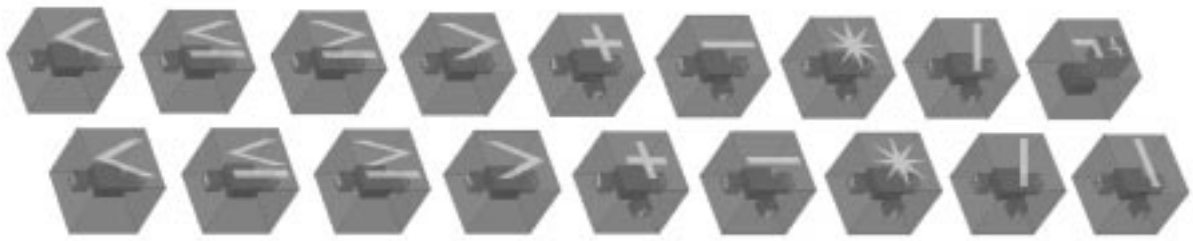


Figure 4.8: Predefined Predicates

indeed present: The initial program contains a "toolkit" of 18 primitive predicate definitions (see Figure 4.8).

As predicates are values, they also have a type associated with them. As said in Section 3.2, we view predicates as functions mapping to propositions. So, each predicate belongs to a *function type*, which is visualized by a *function type cube*. An n-ary function is visualized by the type cube representing the function's domain. This type cube has n ports set into its sides, each one carrying the port icon of the corresponding argument of the function, and filled with the type cube representing the type of this argument.

For example, the type of the integer addition predicate is represented by the type cube shown in Figure 4.7. It consists of a grey opaque cube with the icon "o" on its top, representing the type Prop, and three ports in its side, which carry the same icons as the ports of the addition predicate (see Figure 4.6), and are filled with integer type cubes.

The key difference between function types in Cube and those in functional languages such as ML is that the name (or here, the port icon) of each argument of a function becomes part of its type.

So how can we apply predicate cubes? Assume we want to build a program to convert temperatures between the Celsius and the Fahrenheit scale. Recall that those two scales are related as follows: $F = 1.8 * C + 32.0$ (or $C = \frac{F-32}{1.8}$). We can transform this relation into the logic program

$$\mathsf{conv}(C, F) \Leftarrow \mathsf{times}(C, 1.8, X), \mathsf{plus}(X, 32, F).$$

The program shown in Figure 4.9 is the Cube analog to the above textual logic program. It consists of two empty holder cubes (corresponding to the variables C and F) and two holder cubes filled with floating-point values 1.8 and 32.0, respectively. It also contains a predicate cube referring to the floating-point multiplication predicate, and another predicate cube referring to the floating-point addition predicate. The first port of the multiplication predicate is connected by a pipe to the leftmost empty holder cube, the second one is connected to the holder cube containing the value 1.8, and the third one (the "result") is connected to the first port of the addition predicate. So, if the user puts a value into the leftmost holder cube, the multiplication predicate will receive this value, will multiply it with 1.8, and transfer the result to the addition

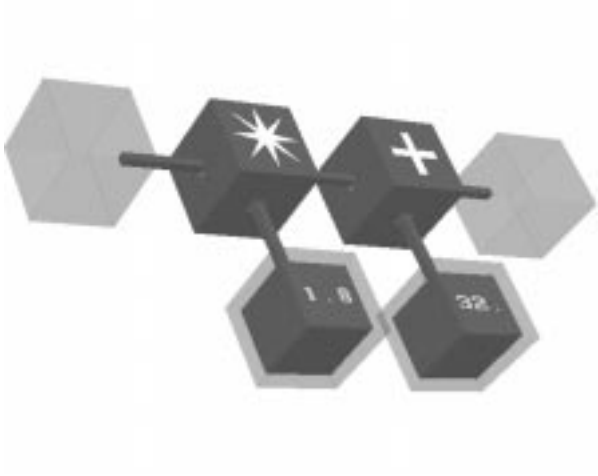


Figure 4.9: Temperature Conversion

Figure 4.10: Converting Celsius to Fahrenheit

Figure 4.11: Converting Celsius to Fahrenheit (Evaluated)

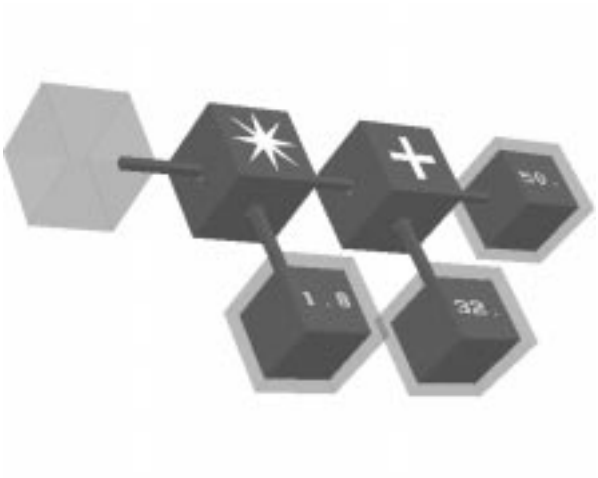


Figure 4.12: Converting Fahrenheit to Celsius

Figure 4.13: Converting Fahrenheit to Celsius (Evaluated)

predicate, which then adds 32.0 to it, and transfers the result to the rightmost holder cube (see Figures 4.10 and 4.11). Alternatively, if the user puts a value into the rightmost holder cube, it will flow into the "result" port of the addition predicate cube, which will now *subtract* 32.0 from it, and transfer the result of this subtraction to the "result" port of the multiplication predicate cube. This cube will *divide* the result of the subtraction by 1.8, and transfer the result of this division to the left holder cube (see Figures 4.12 and 4.13).

This example demonstrates that arithmetic predicates work in either direction. Addition, for instance, can use the first two arguments to produce the third one, or the last two to produce the first one. The "multidirectionality" of predicate applications thus complements nicely the bidirectionality of dataflow in Cube.

Cube is *not* as powerful as a constraint logic language, such as $CLP(\mathcal{R})$ [34] or Janus [72]. It binds variables to values, rather than associating them with constraints. So, in order for an addition to be performed, at least two of its arguments must be known (there is one exception, namely if the first or second argument is 0). As long as not enough arguments are known, the addition is not performed — the predicate is not *resolved*. The semantics of the language (as described in Section 5.3) simply states that such underspecified predicates are not eligible for resolution. The actual implementations (i.e. the CUBE-I and CUBE-II systems), which model goals as concurrent threads, suspend the thread belonging to the underspecified addition until enough data is available.

Figure 4.14: Reporting a Deadlock

There are Cube programs which cannot be "solved" because not enough information is available. This is similar to the situation in $CLP(\mathcal{R})$, where a query can yield three answers: Yes (plus the solution constraints), No, or Maybe. The temperature-conversion program, with neither a Celsius- nor a Fahrenheit-value supplied to it, is such a program. If we try to evaluate it, the system reports that a deadlock occurred: Some threads (supposed to solve a subgoal) are suspended, and there are no threads to wake them up again.

The CUBE-II system reports deadlocks through its solution browser (see Figure 4.14); future implementations might be able to actually highlight the suspended predicate cubes in the program. However, this would require additional run-time information for every thread and cause a performance penalty during evaluation.

4.4 Uninstantiated Variables and Uninstantiated Type Variables

We just mentioned that there is one special case in which the addition predicate can perform a computation, even though only one of its arguments is known (or "*ground*"). This case arises when the value 0 flows into the first or the second argument. The equation $x + 0 = y$ does not allow us to determine the values of x or y, but we know that they must be equal. And equality is the one constraint that even ordinary logic programming languages can handle. Hence, if we evaluate the Cube program shown in Figure 4.15, we would like to learn that the two empty holder cubes must contain the same value.

The question now is how an equality constraint should be visualized. When we *write* a program, we indicate equality constraints between two holder cubes (i.e. unification of two

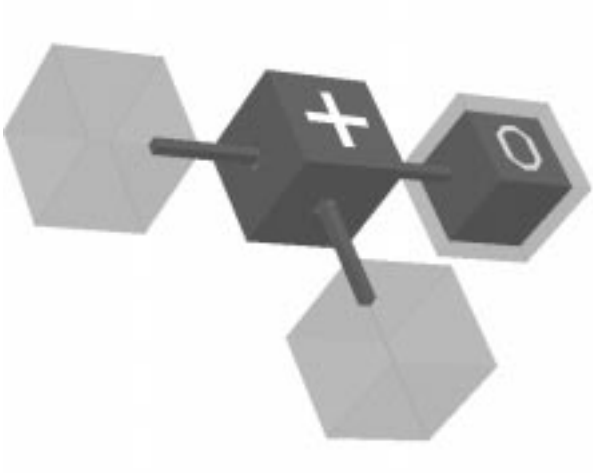


Figure 4.15: Addition With Only One Known Argument

Figure 4.16: Program From Figure 4.15 After Evaluation

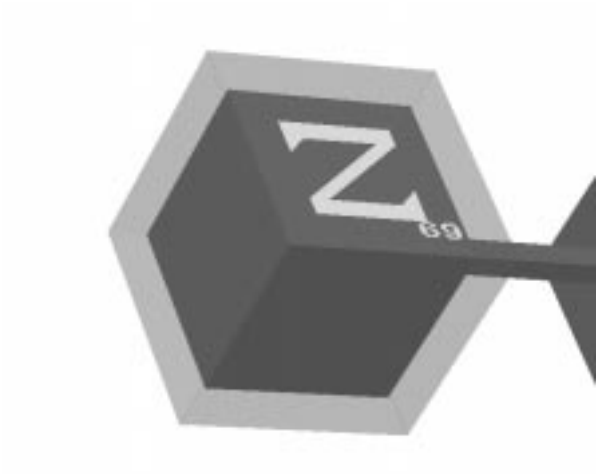


Figure 4.17: Close-Up Onto Left Holder Cube of Figure 4.16

variables) by connecting them through a pipe. So, one solution would be to use the same technique when visualizing results. That is, evaluating the program from Figure 4.15 should create a pipe between the two empty holder cubes.

One of the basic motives for our work on Cube, however, was to use the key ideas of visual data flow languages — with Show and Tell as the prototypical example — to create a visual notation for logic programming. In data flow languages, results of a computation manifest themselves as values flowing into and filling previously empty boxes (variables). Hence, we decided to use the same metaphor — the result of a Cube computation manifests itself through value cubes filling previously empty holder cubes. In retrospect, this decision may have been too conservative — for example, because it barred us from transforming Cube into a visual notation for concurrent constraint logic. It would be interesting to pursue the idea of visualizing equality constraints through pipes, and possibly general constraints between variables through new predicate cubes which appear during a computation. The result might be a language halfway between Show and Tell [37] and Pictorial Janus [35].

Our solution to the problem of visualizing equality constraints follows the tradition of Prolog. In Prolog, new uninstantiated variables are represented as "_n", where n is a unique index, and solving a query which unifies two variables causes them to be bound to the same uninstantiated variable. That is, a Prolog system might behave as follows:

$? \Leftrightarrow X = Y.$

Yes.

$X = \_35$

$Y = \_35$

Cube uses the same idea: an uninstantiated variable is shown as an opaque green cube with a unique index in the lower right corner of its top side. As Cube is a statically typed language, i.e. every variable is associated with a type, we represent this information as well: we show a variable of, say, type Int by superimposing the integer type cube over the indexed opaque green cube. So, in the example from Figure 4.15, the two empty holder cubes both get filled with an opaque green cube with the integer icon "Z" and the index "69" on its top. Figure 4.16 shows this result; Figure 4.17 shows a close-up of the uninstantiated variable.

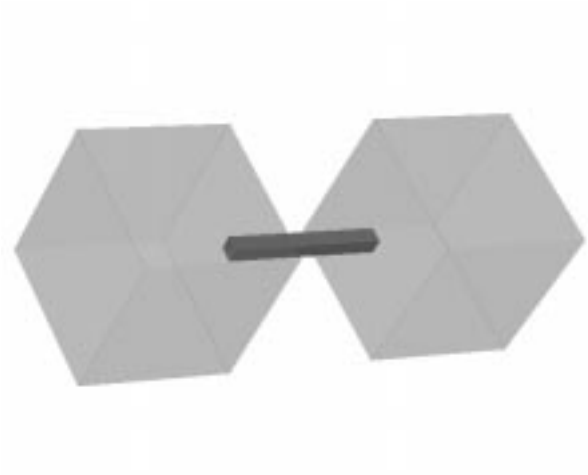


Figure 4.18: Two Empty, Connected Holder Cubes

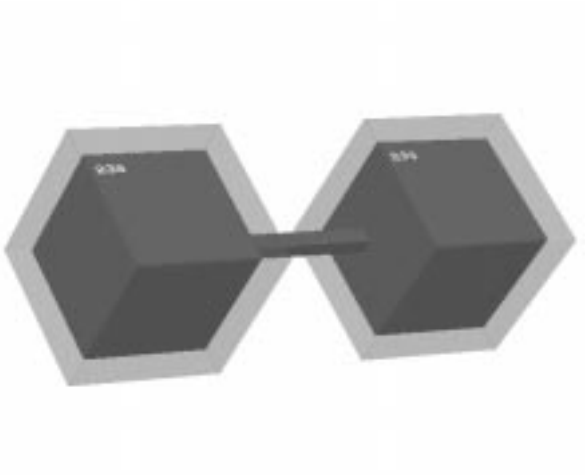


Figure 4.19: Program From Figure 4.18 After Type Inference

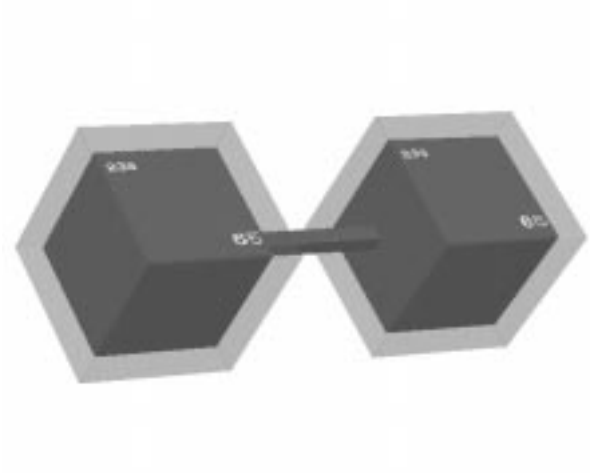


Figure 4.20: Program From Figure 4.18 After Evaluation

Just as there are cases where we know that two holder cubes must contain the same *value*, although we don't know which, there are also cases where we know that they must contain values of the same *type*, although we cannot say what this type should be. Figure 4.18 shows such a case: two empty holder cubes are connected by a pipe. After evaluation, they shall contain the same value, and therefore, they must have the same type as well. But the value could be *anything* of *any type* — the integer 1 just as well as the predicate "plus".

So, we are faced with the problem of visualizing uninstantiated type variables. We adopt the same technique which we used for uninstantiated variables: We represent an uninstantiated type variable by an opaque grey cube with a unique index in the upper left corner of the cube's top side. Figure 4.19 shows the result of performing type inference on the program from Figure 4.18. Both holder cubes contain the same uninstantiated type variable, represented as an opaque grey cube with the index "238" in the upper left corner of its top side.

Figure 4.20 shows the result of evaluating the same program. Both holder cubes now contain the same value, namely the uninstantiated variable number 65 which belongs to the unknown type number 238.

4.5 Predicate Definitions

The part of Cube we have seen so far allows us to combine and connect existing predicates; however, it does not allow us to define new predicates. This ability is crucial in two respects: it provides a mechanism for "procedural abstraction", and it gives us a possibility to perform potentially unbounded computations by allowing us to define *recursive* predicates, predicates which refer to themselves.

4.5.1 A Natural Number Generator

An example of such a predicate would be a natural-number generator: a predicate which generates all the natural numbers, i.e. the integers greater than or equal to 1. In Prolog, we could define such a predicate as follows:

$\mathsf{nat}(1).$

$\mathsf{nat}(X) :\Leftrightarrow \mathsf{nat}(X'), X \text{ is } X' + 1.$

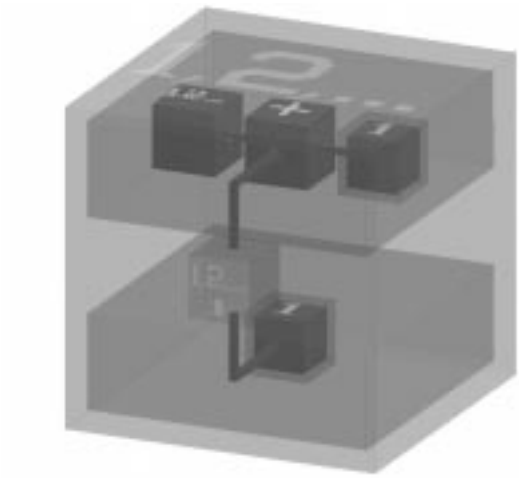


Figure 4.21: A Natural Number Generator

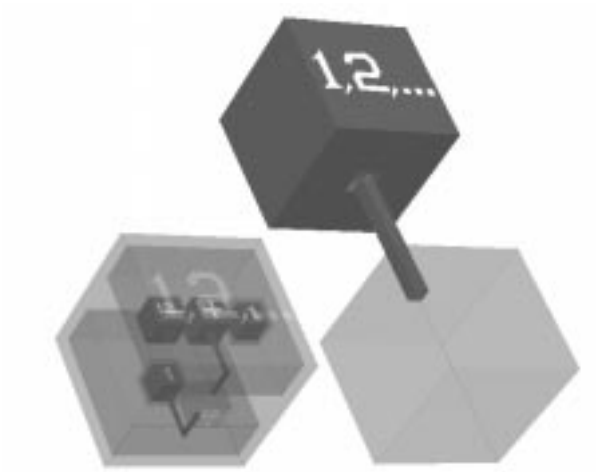


Figure 4.22: Program Computing All Natural Numbers

The query "? $\Leftrightarrow$ nat(X)." would return successive solutions $X = 1$, $X = 2$, $X = 3$, and so on. Note, however, that we cannot use this definition as a *tester*: The query "? $\Leftrightarrow$ nat($\Leftrightarrow$1)." diverges, instead of failing. In order to prevent this, we would have to add an extra subgoal $X > 0$ to the body of the second clause.

Figure 4.21 shows the Cube equivalent of this definition. The outer cube, called a *predicate definition cube*, is a transparent green cube with an icon on its top. This icon provides a name for the new predicate. The small transparent cube set into the center of the front side of the definition cube is a *port*, it represents the formal parameter of the predicate under definition. The icon on its outer side identifies the port.

Inside the predicate definition cube are two transparent boxes, called *planes*. Each plane corresponds to a clause of a textual logic program. Planes are stacked vertically; in Cube, vertical arrangement (in the value world) indicates disjunction, while horizontal arrangement indicates conjunction.

Predicate definition cubes and planes may contain local predicate definition cubes. Predicate definitions occurring at the top level are visible to the entire program (including each other), predicate definitions local to another predicate definition cube are visible to all objects inside this cube, and predicate definitions local to a plane are visible to all objects inside the plane.

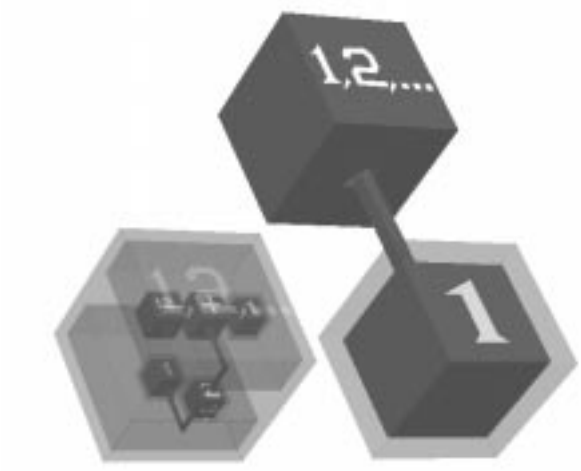


Figure 4.23: First Solution of Program From Figure 4.22

Figure 4.24: Second Solution of Program From Figure 4.22

The lower plane forms the base case of the recursive definition. It contains a holder cube, filled with the value 1, which is connected by a pipe to the port representing the formal parameter.

The upper plane forms the recursive case of the definition. It contains an addition predicate cube, whose first argument is connected to a recursive application of the natural-number predicate, while the second port is connected to a holder cube containing the value 1, and the third argument is connected to the port representing the formal parameter. Note that the recursive case of the natural-number predicate is represented by an opaque cube, i.e. a reference cube. The icon on its top indicates which definition cube it refers to — in this case, the surrounding definition cube. The port in its side carries the same icon as the port of the surrounding definition cube; for predicate cubes which have several parameters, these icons are used to match up actual with formal parameters.

The intuitive meaning of a predicate reference cube is that we could replace it by its corresponding definition cube (after moving the ports around to match them up). This intuition corresponds exactly to what is known as call-by-name semantics in textual programming languages.

If we pose a "query" like the one shown in Figure 4.22, we can imagine that the large reference cube referring to the natural-number predicate gets replaced by the corresponding

predicate definition cube[2]. The value 1 will then flow from the holder cube in the lower plane of the expanded reference cube through the pipe into the port and from there out of the expanded reference cube and into the large empty holder cube. This constitutes the first solution to our query (see Figure 4.23).

We can also imagine that not only the large reference cube got replaced by the definition cube, but that at the same time the recursive reference cube inside the top plane of the expanded reference cube got replaced by the definition cube as well (and the recursive reference cube inside *this* cube as well, and so on ad infinitum). So, the value 1 flows from the holder cube of this second-level expanded reference cube through its port out into a pipe inside the upper plane of the first-level expansion, which takes it to the addition predicate. The addition predicate receives the constant value 1 as a second argument, and returns the value 2, which flows out of its "result" port and through a pipe to the port of the first-level expanded reference cube, and from there into the large holder cube. This constitutes the second solution to the query (see Figure 4.24). It is easy to see how the expansion process can be continued, leading to an infinite number of solutions.

In summary, this example illustrated two key points. One of them is that a Cube query can have multiple solutions (just like a Prolog query). Cube, however, unlike Prolog, explores the paths leading to the various solutions in parallel, and it is guaranteed to find every solution that can be found in finite time (i.e. by a finite number of "expansions"). The second key aspect is that the logical notion of a resolution step — replacing a goal by the subgoals of a matching clause — has an intuitive visual counterpart, namely replacing a reference cube by the corresponding definition cube.

4.5.2 A Factorial Predicate

Figure 4.25 shows another example of a predicate definition cube. This cube defines the factorial predicate. Again, it consists of a transparent green cube, with an icon "!" on its top to name the predicate. Its two ports are set into the left and the right side of the outer cube, and are

[2]In the actual implementation, however, the reference cube remains opaque. Allowing an interpreter to visualize the "expansion" of predicate applications would require additional run-time data structures and imply a performance penalty.

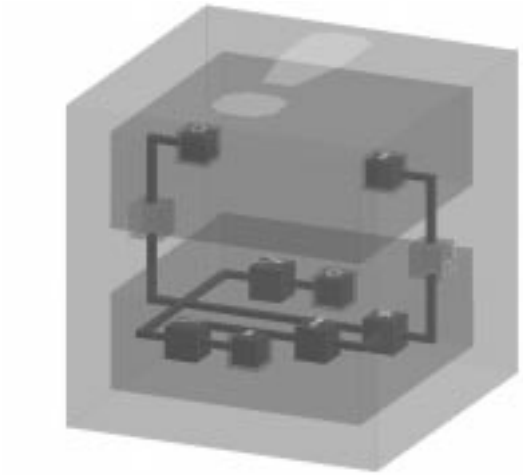


Figure 4.25: Definition of the Factorial Predicate

Figure 4.26: Program Computing the Factorial of 3

Figure 4.27: Program From Figure 4.26 After Evaluation

labeled "n" and "n!". It contains two planes, the upper one representing the base case, and the lower one the recursive case.

The upper plane contains two holder cubes. The left one is filled with the value 0 and connected by a pipe to the left port ("n"); the right one is filled with the value 1 and connected to the right port ("n!").

The lower case contains a comparison predicate, whose two arguments are connected by pipes to the left port ("n") and to a holder cube which contains the value 0. It also contains a subtraction predicate cube, whose two "input" arguments are connected to the left port ("n") and to a holder cube containing the value 1, and whose "output" port is connected by a pipe to the "input" port of a predicate cube which recursively refers to the factorial predicate. The "output" port of the factorial predicate cube is connected to one of the "input" ports of a multiplication predicate cube, whose other "input" port is connected to the left port ("n"), while its "output" port is connected to the right port of the definition cube ("n!").

Now envision a query (like the one shown in Figure 4.26) which contains a predicate cube referring to this definition, and where the user supplies a value, say v, to the left port ("n") of the factorial predicate. Again, we can imagine that the opaque reference cube gets replaced by ("expanded to") the transparent definition cube. The value v flows into the left port, where it splits up, one copy of v flowing through a pipe into the upper plane, and the other copy flowing through the other pipe into the lower plane.

The copy of v which goes to the upper plane flows into a holder cube which already contains the value 0. If v does not unify with 0, then the data flow fails, and with it the entire upper plane. One can imagine that it is simply taken out of the computation. Otherwise, the value 1 contained in the right holder cube flows out through a pipe and into the right port of the expanded reference cube (and possibly into an attached empty holder cube), thereby constituting a solution to the query.

Two remarks are in order here. First, the above intuitive description suggests that the various dataflow operations are performed in some particular sequence, and that data flows into some prescribed direction. While it is often convenient to think about Cube evaluations in such a way, it is also misleading.

Intuitively speaking, all dataflows take place simultaneously, and continue to happen until the system is in equilibrium (has reached a fixed-point). A nice analogy is pipes connecting

containers with different air pressures (different amounts of information) inside. Once the pipes are opened (evaluation is started), air will flow through them until all connected containers have equal pressures. The same equilibrium is reached regardless of the sequence in which the pipes are opened, and the direction of air flow in a pipe is determined only by the pressure difference between two containers.

The second point that should be emphasized is that the result of a unification affects only the computation that happens, logically speaking, within the same conjunction. When the value v entered the port of the expanded reference cube, it got split up into two copies; one went to the upper, the other to the lower plane. A unification that refines v in the upper plane will not affect the copy of v in the lower plane.

Let's get back to our example. As we said before, one copy of the value v flows through a pipe leading into the lower plane, where the pipe branches. One of its ends is connected to one port of a comparison predicate, whose other port receives the value 0. If v is not greater than 0, the comparison fails, and with it the entire plane. The second end of the branching pipe is connected to a subtraction predicate, so v flows into the first argument of this predicate, which receives the value 1 as its second argument. The result of the subtraction, $v \Leftrightarrow 1$, flows out of the third argument port and into the port "n" of a predicate cube recursively referring to the factorial predicate. The result of this computation, $(v \Leftrightarrow 1)!$, then flows out of the "n!" port and into one of the two input ports of a multiplication predicate, its other input port is connected to the third end of the branching pipe, which carries v. The result of the multiplication, $v(v \Leftrightarrow 1)!$, flows finally out of the lower plane and into the port "n!" of the expanded reference cube (and from there possibly into an attached empty holder cube).

Figure 4.27 shows the result of evaluating the query from Figure 4.26.

4.6 Predicates as Values

Cube is a higher-order language, meaning that predicates are first-class values. They can be contained in a holder cube, flow through a pipe, or be argument to another predicate.

Whenever we treat a predicate in such a way — contain it in a holder, transmit it through a pipe, or supply it to another predicate — the ultimate purpose is to eventually *apply* the predicate[3]. So, we need to devise a visual notation for applying this predicate.

Let us look back for a moment to the notation used for an "ordinary" predicate application. A predicate cube is a reference cube, referring to the corresponding predicate definition. We can use a reference cube because the predicate has a name, provided by the definition cube.

We use the same idea to refer to predicate arguments within higher-order predicates. Each port of a predicate definition cube carries an icon, i.e. a name, and we can use this name to refer to the corresponding formal parameter. In other words, the definition cube could contain reference cubes which carry the same icon as one of the outer ports; when the user supplies a value to the port, all the corresponding reference cubes get replaced by this value. We will see an example of this technique in Section 4.8.2.

But how should we apply predicate values for which we don't have a name? For example, how shall we apply a predicate that is supplied through a pipe? The device we employ to do this is called an *application holder cube*, it combines some features of a predicate cube with those of a holder cube. It has ports labeled with icons set into its sides. These ports can be filled with values or connected to pipes. In this respect, it resembles a predicate cube. But unlike a predicate cube, it is neither opaque, nor does it carry an icon that identifies it. Rather, it is transparent like a holder cube, and pipes can be connected to it to supply a value[4].

When an application holder cube receives a value (i.e. a predicate cube) through a pipe, this value will flow inside the holder cube, and its port will be matched up with the ports of the application holder cube (matching is done by icon name).

4.6.1 A Simple Example

Consider the program shown in Figure 4.28. The holder cube on the left side contains the natural-number predicate cube (see Section 4.5.1). It is connected by a pipe to the application holder cube on the right. The application holder cube also contains one port. The icon on the outside of this port is the same as the icon on the port of the natural-number predicate. The

[3] As stated in Section 3.2, Cube does *not* allow us to compare two predicates for semantic equality.

[4] Holder cubes may contain values as well. However, if we were to fill a value (which must be a predicate, i.e. a reference cube or another application holder cube) into an application holder cube, then we could as well omit the application holder cube and instead directly connect to the ports of its value.

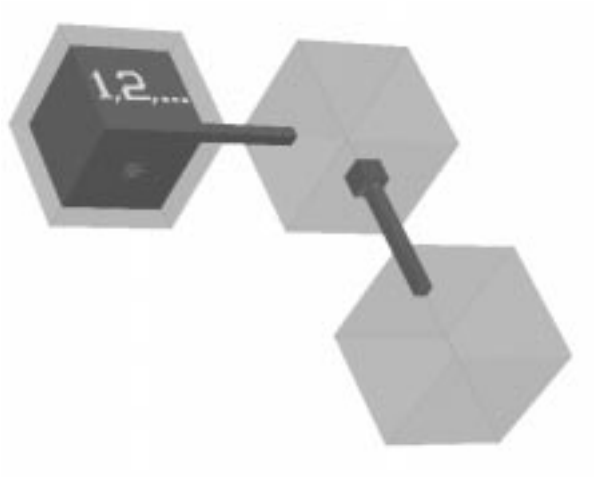


Figure 4.28: Transmitting a Predicate Cube Through a Pipe and Applying It Afterwards (Oblique View)

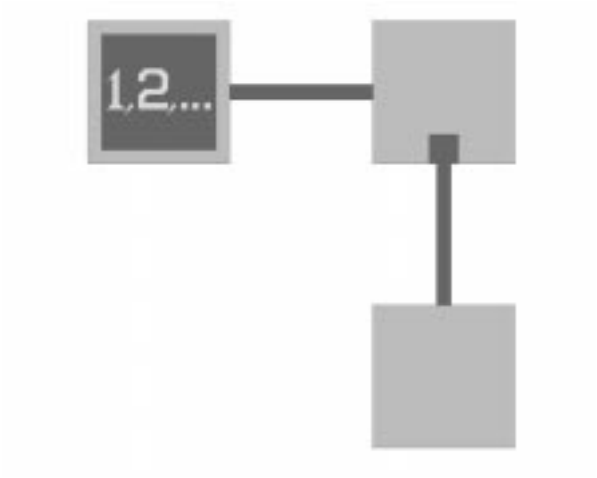


Figure 4.29: Transmitting a Predicate Cube Through a Pipe and Applying It Afterwards (View From Above)

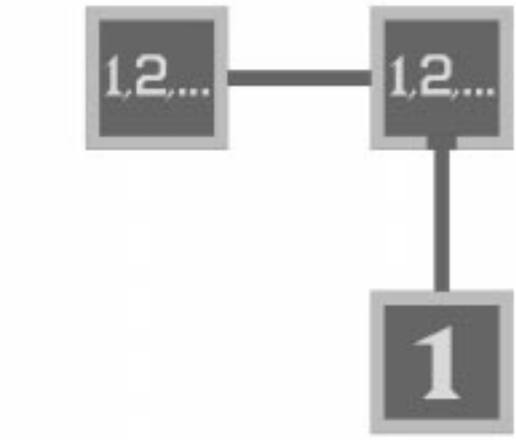


Figure 4.30: Program From Figure 4.29 After Evaluation

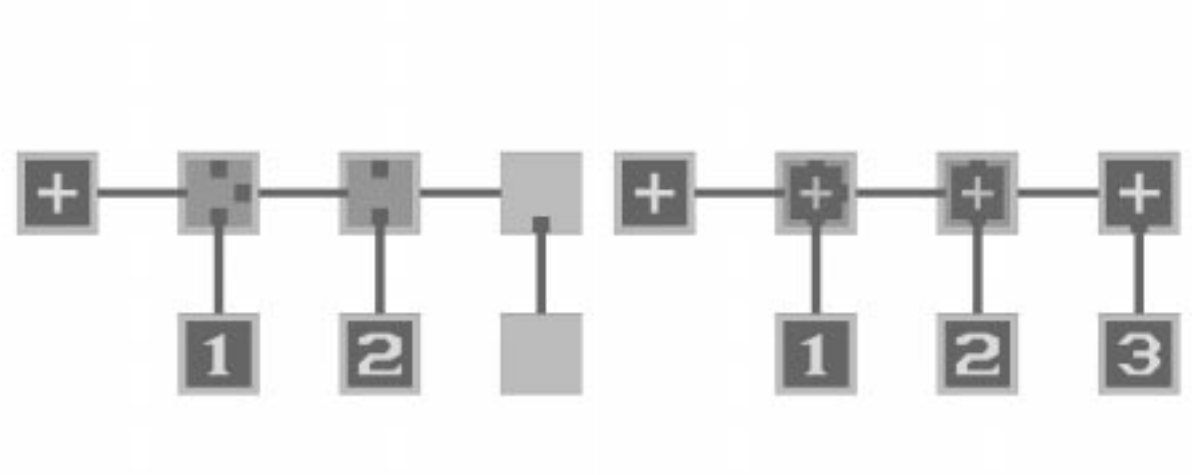


Figure 4.31: Curried Addition

Figure 4.32: Program From Figure 4.31 After Evaluation

port of the application holder cube is connected by a pipe to an empty holder cube in the front right of the picture. Figure 4.29 shows the same program, this time viewed from straight above.

When the user triggers the evaluation of this program, the natural-number predicate cube flows from the left holder cube into the connected application holder cube. Here it gets applied, and starts to produce all the natural numbers. Each natural number flows out of its port (which is matched up with the corresponding port of the application holder cube) and into the empty holder cube on the right. Figure 4.30 shows the first such solution (viewed again from straight abover).

4.6.2 A More Complex Example

Consider the program shown in Figure 4.31. It demonstrates how we can use application holder cubes to achieve the effects of currying, i.e. apply the arguments of a predicate one at a time instead of all together.

The holder cube on the left contains an addition predicate. It is connected by a pipe to an addition holder cube on its right, which has one port with a "first argument of addition" icon on its outside. The port is connected to a holder cube which contains the value 1. The application holder cube is not standing free in the program; it is rather contained inside another holder cube.

We have stated before that enclosing a term cube inside a holder cube corresponds to unifying a term with a variable in a textual logic framework, while an application holder cube corresponds to applying an unknown (i.e. variable) predicate to some argument. So, the textual counterpart of the part of this program we have described so far would be:

$$w = add \wedge x = 1 \wedge w = y \wedge z = y(\mathsf{add}_1 = x) \wedge \cdots$$

where w denotes the left holder cube, x denotes the holder cube which contains 1, y denotes the application holder cube, and z denotes the holder cube around it.

The holder cube containing the application holder cube is connected by a pipe to a second application holder cube on its right. This cube has one port, carrying a "second argument of addition" icon. The port is connected to a holder cube which contains the value 2. The application holder cube is surrounded by a holder cube, which is connected by a pipe to a third application holder cube on its right. This application holder cube is free-standing; it has one port, labeled "third argument of addition", which is connected by a pipe to an empty holder cube below it. So, if we assume that the third application holder cube is denoted by the variable u and the holder cube below it by the variable v, the textual fragment corresponding to this part of the picture is:

$$\cdots \wedge u(\mathsf{add}_3 = v)$$

What happens when this program gets evaluated? The addition predicate cube (or rather the value denoted by it, which is called a *closure* in the textual functional framework) flows from the left holder cube into the application holder cube to its right. Here the value 1 flows into its first port (producing a new closure of arity 2). The result of this application — a cube with two ports left to go — flows from the holder cube into the second application holder cube to its right. Here the value 2 flows into the next port (producing a new closure of arity 1). The result of this application — a predicate cube with one port to go — flows into the third application holder cube, where the port is connected to (i.e. unified with) an empty holder cube. The result of this application is a predicate cube with all ports used up (i.e. a closure of arity 0). The predicate is resolved, and as a result, its third argument gets instantiated to 3. The value of the third argument flows out into the empty holder cube on the right (see Figure 4.32).

Figure 4.33: Renaming the Ports of the Addition Predicate

4.6.3 Renaming of Ports

An application holder cube makes assumptions about the port icons of the predicate that is supplied to it; however, it may receive many different predicates at runtime. Likewise, a predicate (i.e. reference) cube makes assumptions about the port icons of the predicate it refers to; it also may receive many different values at runtime, if the predicate cube refers to a port of a surrounding definition cube instead of directly to a predicate definition. In both cases, the port icons of the reference cube or the application holder cube and the port icons of the value it refers to or contains must match up. In practice, this presents a serious impediment, since it is rare that the port icons of two predicates coincide, although the predicates may well be otherwise of the same type.

We therefore introduce a new device, called a *port renaming cube*. A port renaming cube is a transparent cube (not to be confused with a holder cube) which surrounds a term cube (usually a predicate cube, an application holder cube, or a constructor cube — see below). If we want to rename the port p to p', the port renaming cube carries the icon p' on its transparent hull right above the port labeled p of the term cube inside. Figure 4.33 shows a port renaming cube surrounding an addition predicate cube and renaming all its ports.

4.7 Type Definitions

Cube uses a type system similar to the one used in modern functional languages, such as Miranda [86] or Lazy ML [2]. Similar to those languages, it allows the user to define new types.

Let's take a closer look at type definitions in those textual functional languages. The definition

$$\mathsf{List}\ \alpha\ =\ \mathsf{nil}\ +\ \mathsf{cons}\ \alpha\ (\mathsf{List}\ \alpha)$$

defines two new constructors, nil and cons. A *constructor* is an uninterpreted function symbol. In this particular case, nil is a nullary function (i.e. a simple value) of type "List of α", where α could be any type. nil is conventionally used to denote the empty list. cons is a binary function, which takes a value of type α as first and a value of type "List of α" as second argument, and returns a new "List of α". Again, α ranges over all the types.

Note that the List type is defined in a recursive fashion: the nil constructor forms the base case, the cons constructor is the recursive case, as it creates a list by using another list. It is obvious that every finite list must be terminated by a nil constructor.

The expression cons 1 nil denotes a list whose head is 1, and whose tail is the empty list; or restated, a list which has one element, namely 1. Similarly, the expression cons 1 (cons 2 nil) denotes a two-element list, with 1 as the first and two as the second element.

The expression cons 1 (cons "a" nil) is ill-typed, since cons "a" nil is of type List String whereas 1 is of type Int (we call such a list a *heterogeneous list*).

List is referred to as a *type constructor*, i.e. an uninterpreted function which takes n types as arguments, and returns a new type. So far, we have encountered the nullary type constructors Int, Float, and Prop, and the unary type constructor List.

Now it also becomes clear why Int and Float are distinguished as base types in Cube. The (textual) definition of the integer type would be:

$$\mathsf{Int}\ =\ \cdots\ +\ \text{“}\Leftrightarrow 2\text{”}\ +\ \text{“}\Leftrightarrow 1\text{”}\ +\ \text{“0”}\ +\ \text{“1”}\ +\ \text{“2”}\ +\ \cdots$$

declaring all the integers as constructors of the type Int. But as there are infinitely many integers, such a definition is not possible. The same argument holds for the floating-point type.

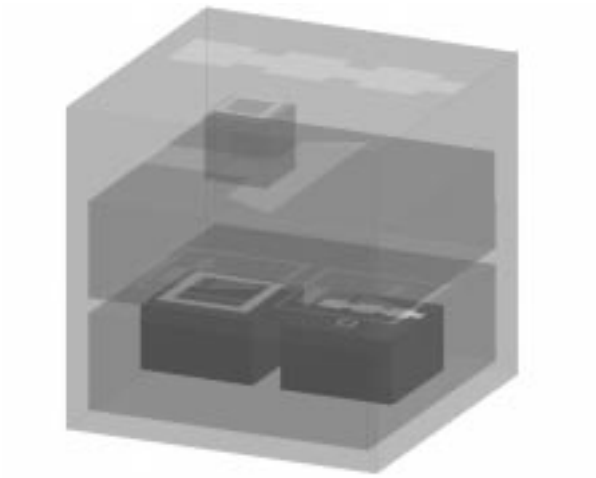


Figure 4.34: List Type Definition

What modifications are needed to adapt this standard notation of type definitions to Cube? Constructors are functions, and as such they have arguments; Cube differs from most functional languages by binding actual to formal parameters not by position, but rather by name (that is, by matching up icons). So, when defining a constructor, we not only need to specify the types of its arguments, but also their icons. By symmetry, we also associate the arguments of type constructors with icons.

Figure 4.34 shows a *type definition cube* which defines the list type. It consists of a grey transparent cube with an icon on its top. The color grey distinguishes types from values, which are green. The icon names the type constructor that is to be defined (List in the textual definition).

The type definition cube has a *port* on its top side, which represents the one formal parameter of the type constructor (α in the textual definition). The port carries an icon on its outside, which is used to distinguish it from other ports.

Inside the type definition cube are two grey transparent boxes, called *type planes*, which represent the two constructors of the list type. The planes are stacked on top of each other; in the context of types, vertical arrangement denotes a type sum, whereas horizontal arrangement denotes a type product. Each plane carries an icon on its top, identifying the constructor. The upper plane, which represents the nil constructor, is empty, as nil is a nullary constructor.

The lower plane represents the cons constructor; it contains two type cubes. The left type cube is a type reference cube, which refers to the port of the enclosing definition cube, i.e. to

α. Note that this notation — referring to a port by using a reference cube which carries the same icon as the port — is used both in the context of types and of values (see Section 4.6). Right above the cube, on the transparent wall of the enclosing box, is an icon (let's call it arg_1) which associates a name with the first argument of the type constructor. The right type cube is a type constructor application. It consists of a type constructor cube, which has a port, and a type contained inside the port.

The type constructor cube is an opaque grey cube with a port set into its top. The icon on top of the cube is the same as that on top of the enclosing definition cube, and the icon labeling the port is the same as that labeling the port of the enclosing definition cube. So, this type constructor is a recursive reference to the type constructor under definition.

The type contained inside the port of this type constructor cube is a type reference cube, referring to the argument of the enclosing type constructor.

This type constructor application cube specifies the type of the second argument of the cons constructor, namely List α. Above the cube, on the transparent wall of the surrounding box, is an icon (let's call it arg_2) which associates a name with this argument.

The two type cubes inside the box are arranged horizontally, so they form a type product; the icon on the box provides a tag for this product, and the vertically stacked planes form a sum of tagged type products.

It should be noted how similar type definition cubes and predicate definition cubes are. Both are represented as transparent cubes with an icon on their top, which names the icon under definition. Ports on their side or top serve as formal parameters. Inside the definition cube are vertically stacked planes, which denote clauses for predicates and variants for types. Vertical arrangement denotes "$\vee$" for predicates and "$+$" for types, while horizontal arrangement denotes "$\wedge$" for predicates and "$\times$" for types.

Type definition cubes can occur in a Cube program wherever predicate definition cubes may occur, and their scope extends just as far. Within this scope, there may be reference (i.e. term) cubes which refer to the constructors defined by the type definition cube. We call such reference cubes *constructor cubes*.

Figure 4.35 shows a constructor cube referring to the cons constructor defined by the list type definition cube from Figure 4.34. Like all (value) reference cubes, it is represented by an opaque green cube. It carries the cons icon on its top, the same icon which is on top of the

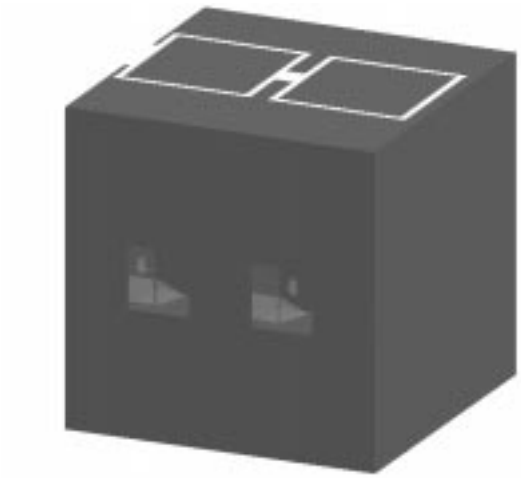


Figure 4.35: The "cons" Constructor

Figure 4.36: The List $[1, 2, 3]$

lower plane defining the cons constructor, and it has two ports in its side, which are labeled by the same two icons as those which are hovering above the two type cubes in the lower plane. The left port can take an argument of any type, say τ, and the right port can take an argument of type "List of τ".

Constructor cubes are first-class values, hence they can be contained in holder cubes, flow through pipes, be passed as arguments to predicates or to other constructors, etc. Their port can be connected to pipes or be filled with values. Whenever we do the latter to build up complex structures, it is customary to move each port so that the icon labeling it occupies the same position as it has in the type plane defining the constructor. The result is a visually pleasing representation of recursive data structures. Figure 4.36 shows the list "$[1, 2, 3]$" enclosed in a holder cube.

This example shows how crucial the choice of constructor icon is for achieving a pleasing representation of recursive structures, and it explains why we chose these particular icons for nil and cons.

Unfortunately, this technique does not always produce visually pleasing data representations. For example, a two-dimensional array would be modeled in Cube as a list of lists. Figure 4.37 shows the standard representation of the array $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. The most intuitive visualization, however, would be to display a two-dimensional grid on top of a cube. Further

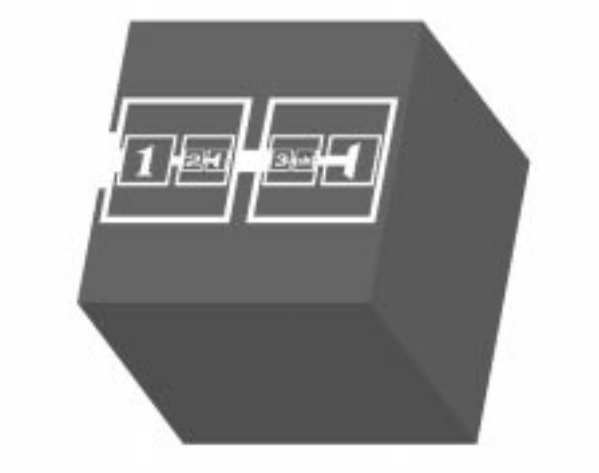


Figure 4.37: Standard Representation of a Two-Dimensional Array

research is needed to devise ways to specify customized visualizations of values, depending on their types.

The program shown in Figure 4.38 shows two holder cubes connected by a pipe. The left holder cube contains a cube representing the list "$[1, 2, 3]$". Upon type inference, the Cube system infers that, since the left holder cube contains a list of integers and the two holder cubes are connected, the right holder cube is also restricted to lists of integers. Therefore it fills the right holder cube with a type cube representing lists of integers: A type reference cube referring to the list type constructor, whose one port is filled with a type reference cube referring to the integer type constructor (see Figure 4.39).

Now consider the program shown in Figure 4.40. Again, it shows two holder cubes connected by a pipe; however, this time the left holder cube contains the value nil, which is of type "List of α", i.e. polymorphic. So, upon type inference, Cube fills the right holder cube with a type reference cube referring to the list constructor, whose port is filled with an uninstantiated type variable (see Figure 4.41).

Constructors are used in Cube — just as in Prolog — both to construct terms and to deconstruct them. We have already seen one way to use constructors to build up new terms: by filling their ports with other term cubes. Alternatively, we can connect them to pipes, which can supply them with values. The same technique is used for deconstruction, except that now values flow *out of* the ports.

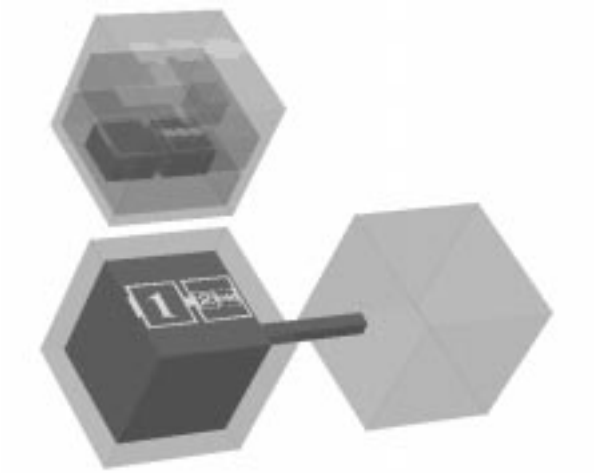


Figure 4.38: List $[1,2,3]$ Flowing Into Empty Holder Cube

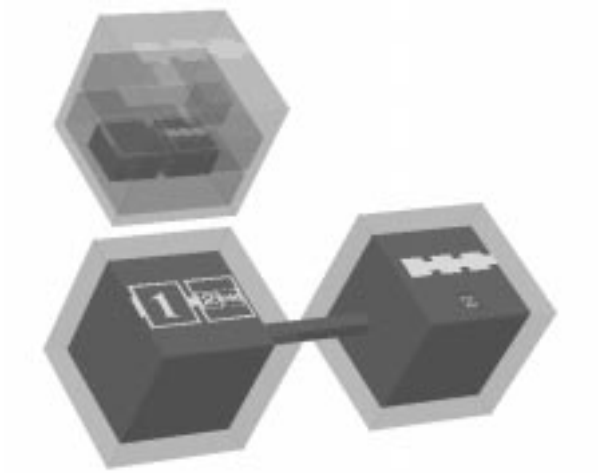


Figure 4.39: Program From Figure 4.38 After Type Inference

Figure 4.40: "nil" Flowing Into Empty Holder Cube

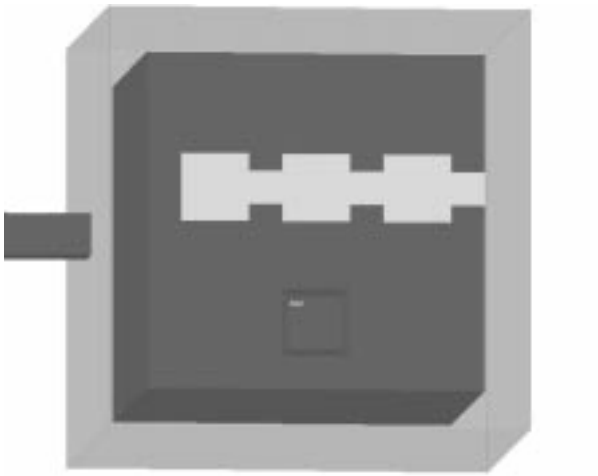


Figure 4.41: Close-Up of Holder Cube From Figure 4.40 After Type Inference

Figure 4.42: Unifying Two Partially Instantiated Structures

Figure 4.43: Program From Figure 4.42 After Evaluation

Indeed, the distinction between constructor and deconstructor is often blurred. Consider the program shown in Figure 4.42. It shows two holder cubes which are both filled with a cons constructor cube, and connected by a pipe. The ports of both constructor cubes are connected via pipes to other holder cubes. One of these holder cubes contains the value 1 and is connected to the "arg_1" port of the constructor cube on the left, another holder cube contains the value nil and is connected to the "arg_2" port of the constructor cube on the right.

Evaluating this program yields the solution shown in Figure 4.43. Note that both constructor cubes served both as constructors and as deconstructors at the same time — one supplied the head of the list and extracted the tail, the other supplied the tail and extracted the head. Note also that the pipe in the center carried data in both directions at once.

4.8 Some Predicates Over Lists

The remainder of this chapter shows some more predicate definitions and typical usages. In particular, we focus on first- and higher-order predicates over lists.

4.8.1 Determining the Length of a List

Consider the predicate definition cube shown in Figure 4.44, which takes two arguments, a list l and an integer n, and holds if the length of l is n.

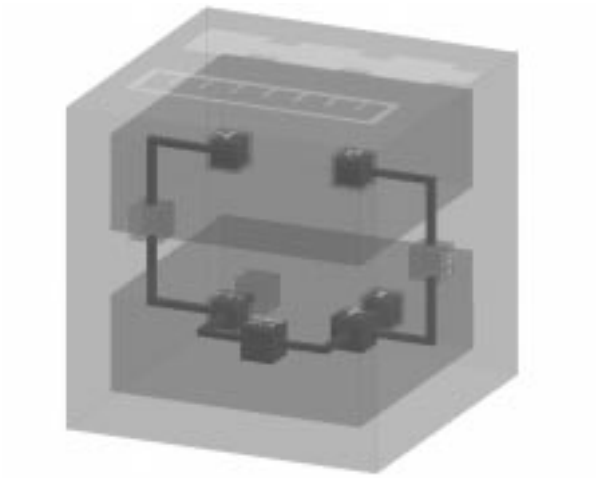


Figure 4.44: Predicate for Computing the Length of a List

The upper plane contains the base case of the definition: the length of the empty list is 0. This is represented by connecting the left port to a holder cube which contains the value nil, and the right port to a holder cube which contains the value 0.

The lower plane contains the recursive case: The length of a non-empty list is 1 plus the length of its tail. This is expressed by connecting the left port to a holder cube which contains a cons constructor. cons is used here to deconstruct the list. Its "arg_1" port (i.e. the head) is connected by a pipe to an empty holder cube, while its "arg_2" port is connected by a pipe to the "list" port of a recursive reference to the length predicate. The "number" port of the length predicate is connected by a pipe to one of the input ports of an addition predicate, whose other input port receives the value 1, and whose output port is connected by a pipe to the right port of the enclosing definition cube.

So, the cons constructor matches an incoming non-empty list, takes it apart, forwards its head to the empty holder cube (i.e. effectively discards it) and its tail to a recursive invocation of the length predicate, which thus determines the length of the tail of the incoming list. The addition predicate adds 1 to this length, yielding the length of the whole incoming list, and sends this value to the right port.

Figure 4.45 shows how this predicate can be used to compute the length of the list "$[1, 2, 3]$"; the solution to this query is shown in Figure 4.46. Interestingly enough, the predicate can also be used with a reversed directionality. Figure 4.47 shows a query that asks for a list of length

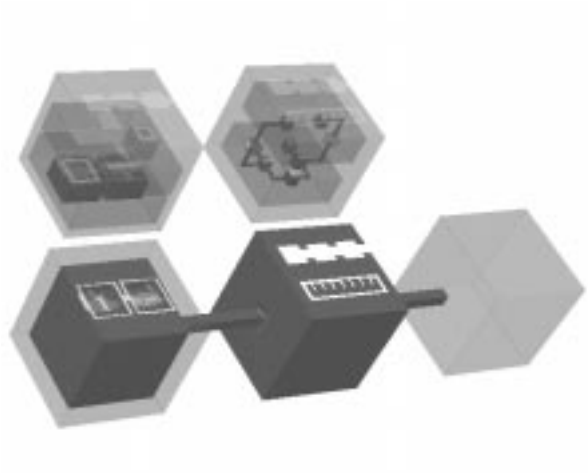


Figure 4.45: Computing the Length of the List $[1, 2, 3]$

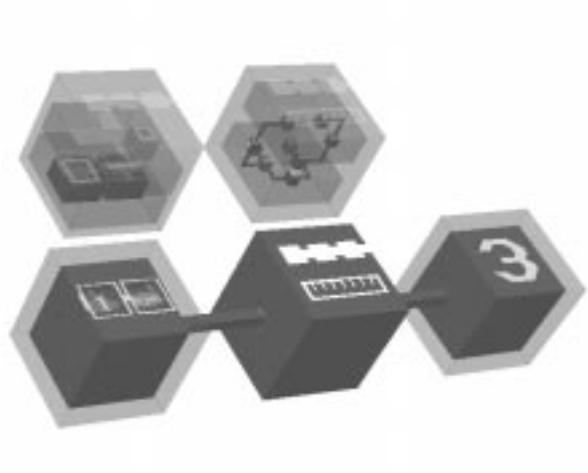


Figure 4.46: Program From Figure 4.45 After Evaluation

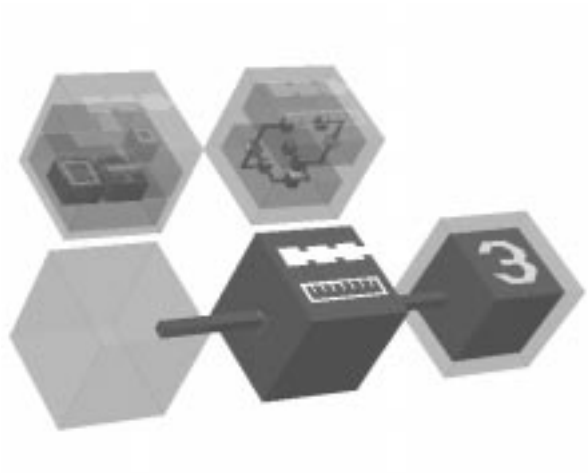


Figure 4.47: Computing a List of Length 3

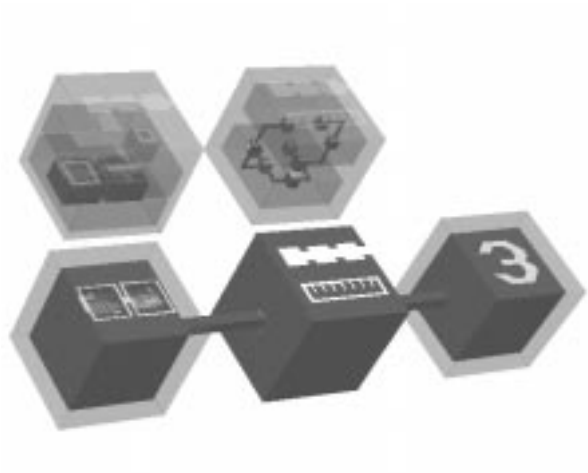


Figure 4.48: Program From Figure 4.47 After Evaluation

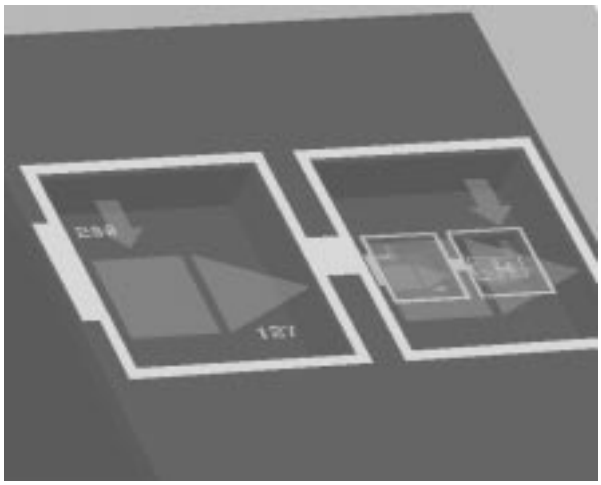


Figure 4.49: Close-Up of Left Holder Cube of Figure 4.48

3, and Figure 4.48 shows the solution to this query: a list with 3 elements, each being a distinct uninstantiated variable, but all of them being of the same, however unknown, type. Figure 4.49 shows a closeup of this list.

4.8.2 Mapping a Predicate Over a List

This example shows the Cube definition of the map predicate which was introduced in Section 3.2. *map* is a higher-order predicate, which takes another (lower-order) binary predicate (say p) and two lists (say $[t_1, \cdots, t_m]$ and $[t'_1, \cdots, t'_n]$), and holds if both lists are of equal length (i.e. if $m = n$) and if the binary predicate holds when applied to corresponding elements in the two lists (i.e. if $p\ t_i\ t'_i$ holds for all $1 \leq i \leq m$).

Figure 4.50 shows the definition cube for this predicate. The port for the lower-order predicate is on the top of the predicate definition cube (which is the convention for predicate arguments), the ports for the two list arguments are on the left and the right.

The upper plane represents the base case: mapping any predicate over the empty list yields the empty list. This is expressed by connecting both "list" ports to holder cubes which contain the value nil.

The lower plane represents the recursive case: both lists are decomposed, the lower-order predicate is applied to their heads, and map is applied recursively to their tails. This is expressed by two holder cubes, one being connected to the left port, the other to the right port, and both

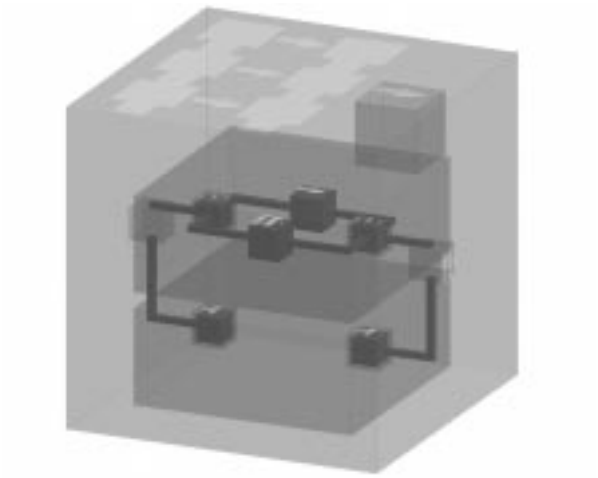


Figure 4.50: The "map" Predicate

being filled with a cons constructor. The "head" and the "tail" ports of both constructors are connected to pipes. The two pipes attached to the "head" ports connect them to the two ports of a reference cube referring to the lower-order predicate; the two pipes attached to the "tail" ports connect them to the two "list" ports of a reference cube referring to map itself, the third port (the predicate argument) of this reference cube is filled with a reference to the lower-order predicate.

Figure 4.51 shows a query which uses the map predicate to map the successor predicate (represented by the definition cube at the top right) over the list "$[1, 2, 3]$". Figure 4.52 shows the solution to this query. The previously empty holder cube on the right is now being filled with the list "$[2, 3, 4]$".

The icons used inside the map predicate definition cube to identify the ports of the lower-order predicate are not the same as the icons identifying the two ports of the successor predicate. Hence, we needed to "wrap" the successor reference cube into a port renaming cube before supplying it to the map predicate.

Note that the "predicate" argument of map must always be completely ground; otherwise, the evaluation of the recursive case in which it is used will suspend until the predicate is ground. The allowable instantiation patterns of the two "list" arguments, on the other hand, depends only on the predicate argument. If we use a predicate argument which expects both of its arguments to be ground (such as "greater"), then both list arguments of map have to be completely ground; otherwise, the evaluation suspends. If we use a predicate argument which

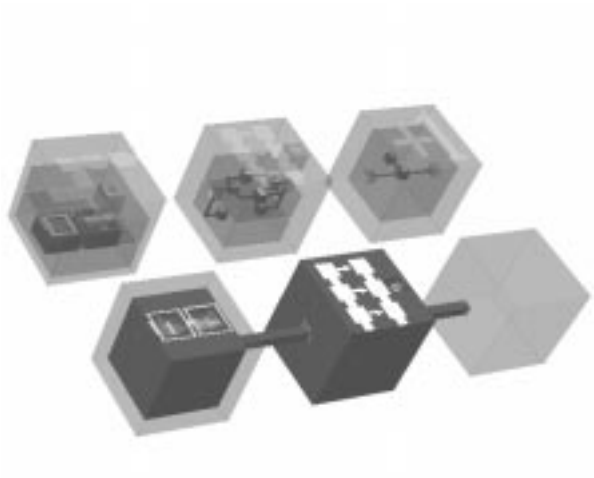


Figure 4.51: Mapping the Successor Predicate Over the List $[1, 2, 3]$

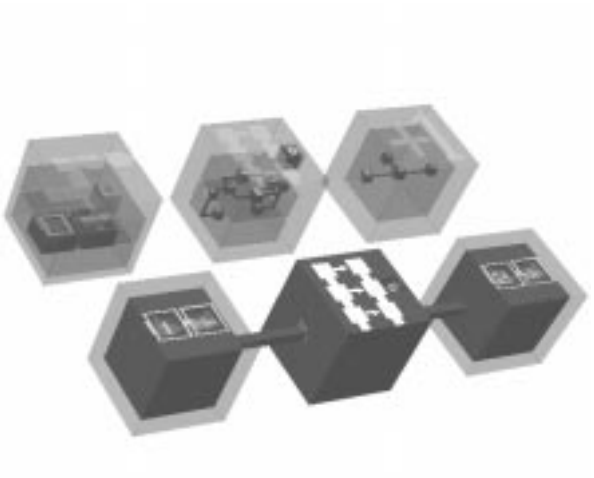


Figure 4.52: Program From Figure 4.51 After Evaluation

expects at least one of its arguments to be ground, then for each two corresponding elements of the two lists, at least one has to be ground. Finally, if we use a predicate argument which expects neither of its arguments to be ground or even instantiated (such as "equal"), then the two list arguments do not have to be instantiated at all, instead, map will generate all possible solutions.

4.8.3 Filtering Out Some Elements of a List

filter is a higher-order function frequently used by functional programmers. Axiomatically, it is defined as follows:

$$\textit{filter}\ p\ [e_1, \cdots, e_n] \equiv [e_{i_1}, \cdots, e_{i_k}]$$

$$\text{where}\ \ 1 \leq i_1 < \cdots < i_k \leq n\ \ \text{and for all}\ j \in \{1, \cdots, n\} :\ p\ e_j\ = \begin{cases} \text{true} & \text{if } j \in \{i_1, \cdots, i_k\} \\ \text{false} & \text{otherwise} \end{cases}$$

So, *filter* takes a function p and a list $[e_1, \cdots, e_n]$. p is a function which maps a value of type α to a boolean, and the e_i are of type α. It returns a list of all those e_i for which p "holds" (i.e. returns **true**).

In Lazy ML[5], we could define *filter* as follows:

[5]Lazy ML in some way resembles Prolog, as it allows function definitions to be split into several cases (a weaker form of clauses) and uses pattern matching (a weaker form of unification) to bind actual to formal parameters.

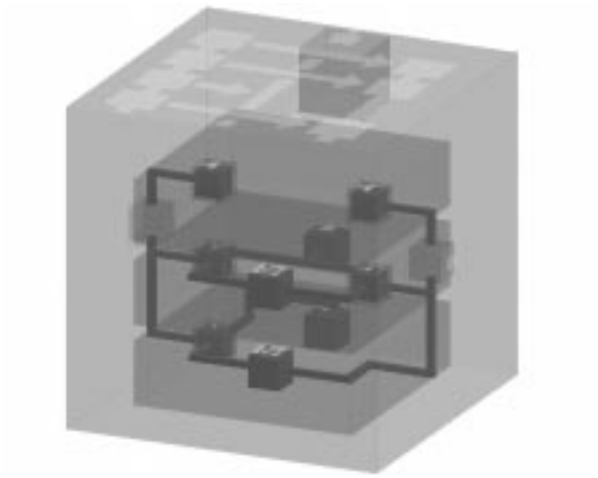


Figure 4.53: The "filter" Predicate

$$\textit{filter}\ p\ [\,]\ =\ [\,]$$
$$\textit{filter}\ p\ (h\,.\,t)\ =\ \textbf{if}\ p\ h\ \textbf{then}\ t\,.(\textit{filter}\ p\ t)\ \textbf{else}\ \textit{filter}\ p\ t$$

Translating this function into a predicate gives rise to three clauses: one for the base case, one for the recursive case where the predicate holds for the head of the list, and one for the case where it does not hold. This third case requires the use of negation to explicitly state that the predicate may not hold. Here is the Prolog definition of filter:

$$\mathsf{filter}(P, [\,], [\,]).$$
$$\mathsf{filter}(P, X\,.\,L, X\,.\,L') \ :\Leftrightarrow Q\ =..\ [P, X], \mathsf{call}(Q), \mathsf{filter}(P, L, L').$$
$$\mathsf{filter}(P, X\,.\,L, L') \ :\Leftrightarrow Q\ =..\ [P, X], \mathsf{not}(Q), \mathsf{filter}(P, L, L').$$

It should be mentioned that Prolog's not is a rather problematic metapredicate, as it does not capture the semantics of negation correctly. For instance, the query

$$?\Leftrightarrow \mathsf{not}(\ X = 0\ ), X = 1.$$

fails, although logically it should succeed. This "unlogical" behavior is caused by the fact that not tried to decide on the validity of "$X = 0$" before X was ground. In Cube, negation is suspended until the negated atomic formula is completely ground.

Figure 4.53 shows the *filter* predicate definition cube. The port on its top takes the lower-order predicate (the "tester"), the left port takes the incoming list, and the right port returns the filtered list. Inside the cube are three planes.

The top plane represents the base case: Filtering the empty list produces the empty list. This is expressed by connecting both "list" ports to holder cubes which contain the value nil.

The second plane represents the recursive case in which the tester holds. The left port is connected to a holder cube which contains a cons constructor cube. The "tail" port of the constructor is connected by a pipe to the "input list" port of a reference cube which recursively refers to the *filter* predicate, while the "head" port of the constructor is connected to the port of a reference cube referring to the "tester" predicate (by using the same icon as the "predicate" port of the enclosing definition cube). The pipe connecting the "head" port to the tester has a T-joint, its third end leads to the "head" port of a second cons constructor cube, whose "tail" port is connected to the "result list" port of the filter predicate. This constructor cube is contained in a holder cube, which is connected by a pipe to the "result list" port of the enclosing definition cube.

Finally, the lower plane represents the recursive case in which the tester does not hold. The left port of the enclosing definition cube is connected to a holder cube which contains a cons constructor cube. Again, the "tail" port of the constructor is connected to the "input list" port of a recursive reference to the *filter* predicate, while the "head" port is connected to the port of the tester predicate. But in this plane, the tester predicate is contained inside the port of a negation predicate cube. The "result list" port of the recursive reference to the filter predicate is connected by a pipe to the "result list" port of the enclosing definition cube.

Figure 4.54 shows the filter predicate being applied to the lower-order predicate *odd* and to the list "$[1,2,3]$". Figure 4.55 shows the same query after evaluation. The previously empty holder cube on the right has been filled with the list "$[1,3]$".

Note that both the predicate argument to *filter* and the "input" list argument have to be ground: the negation predicate expects its argument to be completely ground, and this argument is the lower-order predicate applied to each of the elements of the input list in turn. So, the filter predicate is truly unidirectional.

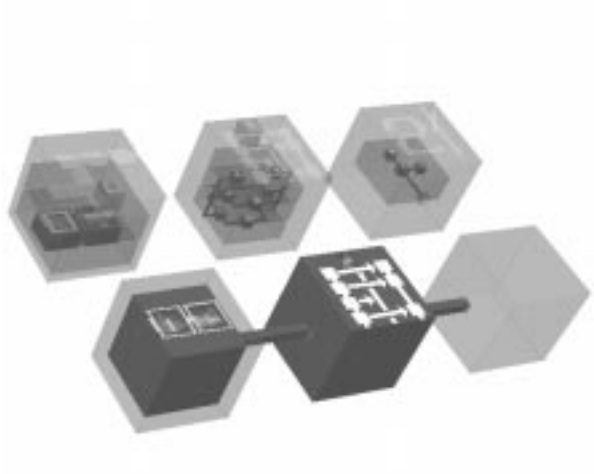


Figure 4.54: Filtering all but the odd numbers from the list $[1, 2, 3]$

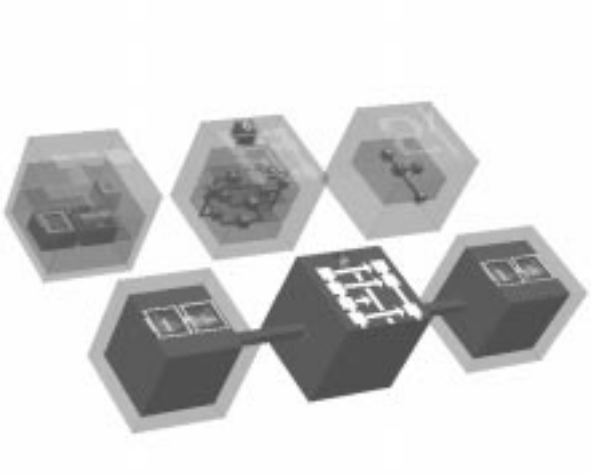


Figure 4.55: Program from Figure 4.54 after evaluation

Chapter 5

Formal Description

This chapter develops a formal definition of Cube. The way we do this is rather indirect. Instead of giving a semantics based directly on pictures, we first describe a translation algorithm from Cube pictures into a textual language, and then give type inference rules and an operational (rewrite) semantics for this textual language. The understanding is that a Cube program is well-typed if it translates into a well-typed textual program, and that a Cube program yields a particular (visual) result if it translates into a textual program which yields a textual result that is the translation of the visual result.

The structure of this section is as follows: Section 5.1 develops a translation scheme from Cube into a textual language L_0, Section 5.2 describes a type system for L_0 (or a derivative of it), and Section 5.3 defines a rewrite system which takes expressions of (a derivative of) L_0 into a normal form.

5.1 Translation From Pictures to Text

This section gives translation rules from Cube pictures into words of a textual language L_0. Table 5.1 gives the context-free syntax of L_0.

In our textual language, the nonterminal symbols P, k, K, p, π, i, and f refer to predicate, constructor, typeconstructor, port, and typeport symbols, and integer and floatingpoint constants. For convenience, we will use the symbols $\boxed{P}$, $\boxed{k}$, $\boxed{K}$, $\boxed{p}$, $\boxed{\pi}$, $\boxed{i}$, and $\boxed{f}$ to refer to predicate, constructor, typeconstructor, port, typeport, integer and floatingpoint icons in

E	$\in$	Program	
c	$\in$	Conjunction	
d	$\in$	Disjunction	
t	$\in$	Term	
D	$\in$	Definition	
x	$\in$	Variable	
k	$\in$	Constructor $\subset$ Variable	
P	$\in$	Predicate $\subset$ Variable	
i	$\in$	Integer	
f	$\in$	Floatingpoint	
p	$\in$	Portname	
K	$\in$	Typeconstructor	
α	$\in$	Typevariable	
V	$\in$	Typevariant	
π	$\in$	Typeportname	
τ	$\in$	Type	

E	::	$\Leftarrow c$	*Program*
d	::	**letrec** $D_1, \cdots, D_m$ **in** $c_1 \vee \cdots \vee c_n$	*Disjunction*
c	::	**letrec** $D_1, \cdots, D_k$ **in** $\exists x_1, \cdots, x_m . t_1 \wedge \cdots \wedge t_n$	*Conjunction*
t	::	i	*Integer constant*
	$\mid$	f	*Floating-point constant*
	$\mid$	x	*Variable*
	$\mid$	$t_1 = t_2$	*Unification*
	$\mid$	$t_1(p = t_2)$	*Application*
	$\mid$	$t\ (p \rightarrow p')$	*Parameter renaming*
D	::	**type** $K\{\pi_1 = \alpha_1, \cdots, \pi_m = \alpha_m\} = V_1 + \cdots + V_n$	*Type definition*
	$\mid$	**pred** $P = \lambda\{p_1 = x_1, \cdots, p_k = x_k\}.d$	*Predicate definition*
V	::	$k\ \{p_1 : \tau_1, \cdots, p_n : \tau_n\}$	*Type variant*
τ	::	α	*Type variable*
	$\mid$	$K\ \{\pi_1 = \tau_1, \cdots, \pi_n = \tau_n\}$	*Type constructor application*
	$\mid$	$\{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau$	*Function type*

Table 5.1: Syntax of L_0

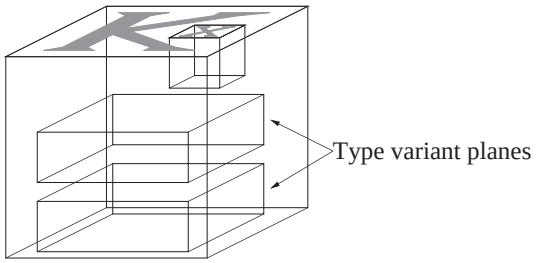


Figure 5.1: Visual Syntax of Type Definition Cubes

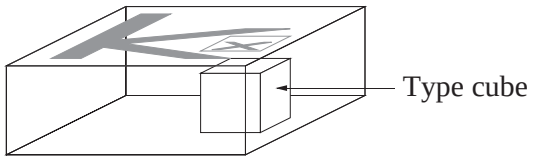


Figure 5.2: Visual Syntax of Type Variant Planes

the visual language. The understanding is that $\boxed{x}$ and x refer to two different representations of the same symbol.

Concrete predicate, constructor, typeconstructor, port, and typeport symbols of the textual language are represented by strings. By convention, we start typeconstructor names (such as *Int* or *List*) with an uppercase letter, and predicate and constructor names (such as *plus* or *cons*) with a lowercase letter. Predicates and constructors are both special kinds of variables. In addition, there is a sequence $x_1, x_2, x_3, ...$ of variable names used to obtain new variables. Analogously, there is a sequence $t_1, t_2, t_3, ...$ of type variable names used to obtain new type variables. By convention, we use a slanted font to refer to type constructors and type variables, and predicates, constructors and other variables.

We use subscripted strings to refer to the parameters of type constructors, predicates, and constructors. For example, List_1 refers to the first parameter of the *List* type constructor, and plus_3 refers to the third parameter of the *plus* predicate. In Cube, ports are *not* first-class values, so port symbols will always be constants, which we indicate by using a *sans-serif* font.

A *type definition cube* has the form shown in Figure 5.1. It consists of a transparent grey cube with icon $\boxed{K}$, ports with icons $\boxed{\pi_1}, ..., \boxed{\pi_m}$ $(m \geq 0)$, and n type variant planes $(n > 0)$,

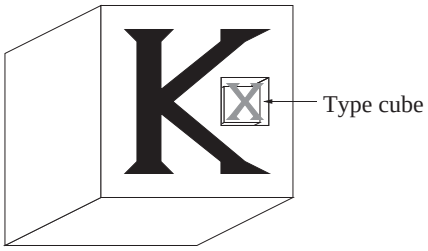


Figure 5.3: Type Constructor Application

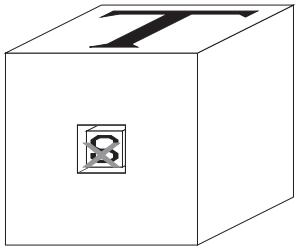


Figure 5.4: Function Type

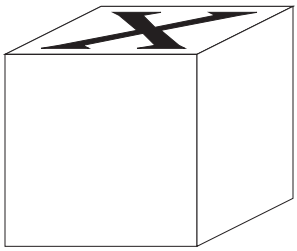


Figure 5.5: Type Reference Cube

which represent the n variants of the sum-type defined here. By convention, we place the type ports always on top of the type naming cube.

A *type variant plane* has the form shown in Figure 5.2. It consists of an enclosing grey plane, carrying a transparent, grey constructor icon $\boxed{k}$ on its top, and n type cubes inside the plane ($n \geq 0$). The part of the plane above each type cube carries a transparent, green port icon $\boxed{p_i}$ ($1 \leq i \leq n$).

A *type cube* can have one of three forms:

1. A *type constructor application cube*, as shown in Figure 5.3, consists of an opaque, grey cube with a type constructor icon $\boxed{K}$ on its top, and ports with port icons $\boxed{\pi_1}$, ..., $\boxed{\pi_n}$ ($n \geq 0$) set into the top. Each port must be filled with a type cube. In order for this expression to be well-formed, it must be within the scope (see page 49) of a type definition cube (i.e. a naming cube) defining a type constructor $\boxed{K}$ with exactly n ports labeled $\boxed{\pi_1}$, ..., $\boxed{\pi_n}$.

2. A *function type cube* of the form shown in Figure 5.4. It consists of a (non-function) type cube, called the *result type cube*, and ports labeled $\boxed{p_1}, ..., \boxed{p_n}$ set into its side. Each port must be filled with a type cube, called an *argument type cube*.

3. A *type port reference cube* of the form shown in Figure 5.5 is an iconic reference to a type parameter. It consists of an opaque, grey cube with a type port icon $\boxed{\pi}$ on top. In order to be well-formed, the enclosing type definition cube must have a port named $\boxed{\pi}$.

At this point, a small digression is in order. Initially, we wanted the syntax of type definitions and predicate definitions to be as similar as possible. Therefore, we included *type holder cubes* (representing type variables) and *type pipes* (representing the unification of type variables) into the language. These constructs mirrored the holder cube and pipe constructs of the value-denoting fragment of the language, and thus provided a nice symmetry. On the other side, they added a significant layer of complexity to the translation scheme for types, since unification of types had to be performed statically, and could potentially fail, indicating a syntactically ill-formed program. For this reason, we omitted type holder cubes and type pipes from the final version of the language. The initial version of the language and the more complicated translation scheme are described in [55].

Type definition cubes are translated into their textual representation by the following rules:

1. A *type definition cube* with type constructor icon $\boxed{K}$, type ports labeled $\boxed{\pi_1}, ..., \boxed{\pi_m}$, and variants translating to $V_1, ..., V_n$ translates to

$$\textbf{type}\ K\{\pi_1 = \alpha_1, \cdots, \pi_m = \alpha_m\} = V_1 + \cdots + V_n$$

where $\alpha_1, ..., \alpha_m$ are new and distinct type variables. We say that π_i *is associated with* α_i $(1 \leq i \leq m)$.

2. A *type variant plane* with constructor icon $\boxed{k}$, and type cubes translating to $\tau_1, ..., \tau_n$ under port icons $\boxed{p_1}, ..., \boxed{p_n}$ translates to

$$k\ \{p_1 : \tau_1, \cdots, p_n : \tau_n\}$$

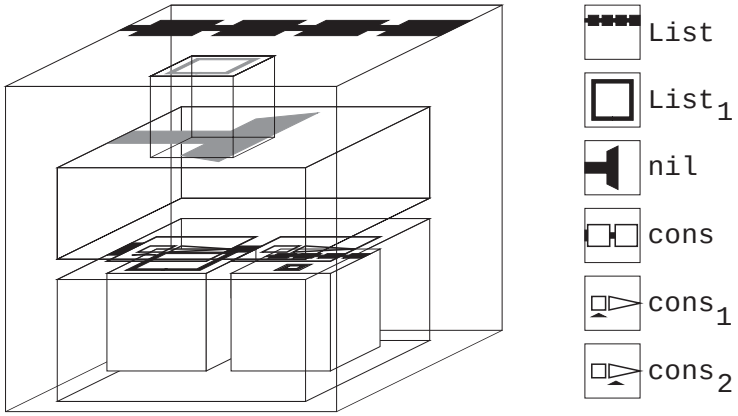


Figure 5.6: Definition of a List Type

3. A *type constructor application cube* with type constructor name $\boxed{K}$ and type ports $\boxed{\pi_1}$, ..., $\boxed{\pi_n}$, where the holder cubes of the ports translate to $\tau_1, ..., \tau_n$, translates to

$$K\ \{\pi_1 = \tau_1, \cdots, \pi_n = \tau_n\}$$

4. A *type port reference cube* with type port icon $\boxed{\pi}$, referring to the type parameter π associated with the type variable α, translates to α.

5. A *function type cube* with argument type cubes translating to $\tau_1, ..., \tau_n$ in ports labeled $\boxed{p_1}$, ..., $\boxed{p_n}$ and a result type cube translating to τ, translates to

$$\{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau$$

At this point, we have a representation D^T of the type definition cube. D^T is of the form **type** $K\{\pi_1 = \tau_1, ..., \pi_m = \tau_m\} = V_1 + ... + V_n$.

Example 5.1.1 (Translation of the list type definition) The type definition cube shown in Figure 5.6, which defines the list type, consists of a type definition cube with icon *List*, which has one port, labeled List_1, set into its top. Inside the type definition cube are two type variant planes, the upper one carrying the type constructor icon *nil* and being empty, and the lower one carrying the type constructor icon *cons*, and being filled with two type cubes below port icons cons_1 and cons_2. One of them is a type reference cube (call it τ_1) with icon List_1 on its

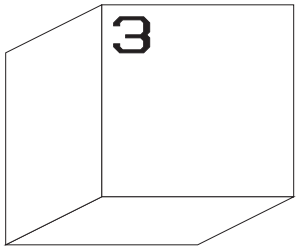


Figure 5.7: Visual Syntax of Type Variables

top (thus referring to the port of the type definition cube). This type cube is below the port icon cons_1. The other is a type constructor application cube (call it τ_2), consisting of an opaque cube with icon *List* on its top, and a port labeled List_1 set into it. The port contains another type reference cube (call it τ_3), also with icon List_1 on its top.

The type definition cube is translated as follows: We associate the type port icon List_1 with a type variable t_1 (by rule 1). The type reference cube τ_1 carries the icon List_1 and thus translates to t_1 (by rule 4). The same rule applies for the type reference cube τ_3. The type constructor application cube τ_2 then translates to $List\{\mathsf{List}_1 = t_1\}$ (by rule 3). The lower type variant plane now translates to $cons\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : List\{\mathsf{List}_1 = t_1\}\}$ and the upper plane simply to *nil* (by rule 2). The type definition cube finally translates (by rule 1) to the representation D^T

$$\textbf{type}\ List\{\mathsf{List}_1 = t_1\} = nil + cons\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : List\{\mathsf{List}_1 = t_1\}\}$$

Section 5.2 describes a type inference system which determines if an expression is well-typed, and computes the type of each subexpression. Some of these types will be displayed right inside the Cube program. In particular, the type of each empty holder cube (including empty ports) is indicated by filling the holder cube with a type cube.

An inferred type can be a type variable, a type constructor application, or a function type. The corresponding type cubes are constructed using the following three rules:

1. a *type variable* t_i (where i indicates the position of t_i in the sequence of type variables) is represented by an opaque grey cube with a grey icon on its top, such that the number

i in the top left corner of the (otherwise blank) icon. Figure 5.7 shows the representation of the type variable t_3.

2. A *type constructor application* $K\ \{\pi_1 = \tau_1, \cdots, \pi_n = \tau_n\}\ \ (n \geq 0)$ is represented by a type constructor application cube: an opaque grey cube with icon $\boxed{K}$ and ports named by transparent grey icons $\boxed{\pi_1}, ..., \boxed{\pi_n}$ set into the cube's top, and filled with the representations of $\tau_1, ..., \tau_n$.

3. A *function type* $\{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau\ \ (n > 0)$ is represented by a function type cube: a type cube representing τ, with ports named $\boxed{p_1}, ..., \boxed{p_n}$ set into its sides and filled with the representations of $\tau_1, ..., \tau_n$.

This concludes the translation schemes for types and type definitions. The remainder of this section specifies schemes for translating value-denoting pictures into L_0 expressions, and vice versa. The first few definitions are aimed at resolving the pipes in a picture:

Definition 5.1.1 Let x be a variable and t be a term. A *constraint* is an equation of the form $x = t$.

Definition 5.1.2 Each holder cube in a Cube program (including the ports of predicate definition cubes and of application cubes) is *associated* with a variable, such that there is an injective (1-1) mapping from holder cubes to variables.

Definition 5.1.3 A holder cube is called a *top-level holder cube* if it is inside a conjunction (see below), with no cube inside this conjunction surrounding it. Otherwise, it is called a *lower-level holder cube*.

Icons are visible only within a certain scope, namely the box enclosing the naming cube by which they are defined. In order to deal with the issue of scope, we need to establish some more definitions, leading up to the notion of an *icon environment*.

Definition 5.1.4 An *icon binding* is an association $\boxed{x} \mapsto x$ between an icon $\boxed{x}$ and a variable x.

Definition 5.1.5 An *icon environment* $\rho = \{\boxed{x_1} \mapsto x_1, ..., \boxed{x_n} \mapsto x_n\}\ \ (n \geq 0)$ is a set of icon bindings, i.e. a mapping from icons to variables. ρ_0 denotes the *initial icon environment*,

the icon environment which maps the icons of the predefined predicates and constructors to variable names.

Definition 5.1.6 Let ρ be an icon environment and $\boxed{x}$ be an icon. Then $\rho(\boxed{x})$ is defined to be x if there is a binding $\boxed{x} \mapsto x$ in ρ. Otherwise, $\rho(\boxed{x})$ is undefined.

Definition 5.1.7 Let ρ be an icon environment and $\boxed{x} \mapsto x$ be a binding. Then $\rho[\boxed{x} \mapsto x]$, the *extension of* ρ *by* $\boxed{x} \mapsto x$, is defined as follows:

$$(\rho[\boxed{x} \mapsto x])(\boxed{x'}) = \begin{cases} x & \text{if } \boxed{x'} = \boxed{x} \\ \rho(\boxed{x'}) & \text{otherwise} \end{cases}$$

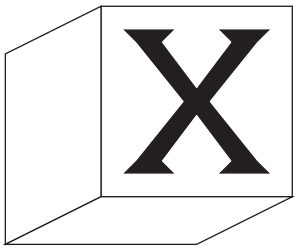
$\rho[\boxed{x_1} \mapsto x_1, ..., \boxed{x_n} \mapsto x_n]$ is an abbreviation of $(...(\rho[\boxed{x_1} \mapsto x_1])...)[\boxed{x_n} \mapsto x_n]$.

Definition 5.1.8 (Yielding of Bindings)

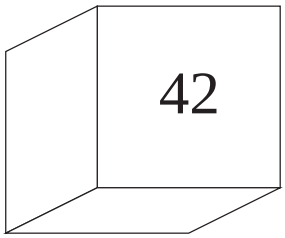
1. A type variant plane with icon $\boxed{k}$, which translates to $k\{x_1 : \tau_1, ..., x_n : \tau_n\}$, yields the bindings $\boxed{k} \mapsto k$.
2. A type definition cube, whose n type variant planes yield the bindings $\boxed{k_1} \mapsto k_1, ..., \boxed{k_n} \mapsto k_n$, yields the bindings $\boxed{k_1} \mapsto k_1, ..., \boxed{k_n} \mapsto k_n$.
3. A predicate definition cube with predicate icon $\boxed{P}$, which translates (as described below) to **pred** $P = e$, yields the binding $\boxed{P} \mapsto P$.

There are six kind of term cubes: reference cubes, integer cubes, floatingpoint cubes, holder cubes, application cubes, and port renaming cubes.

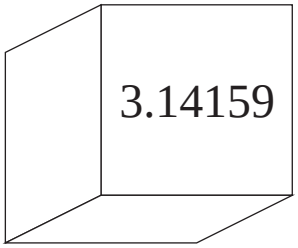
- A *reference cube* is of the form shown in Figure 5.8.a. It consists of an opaque green cube with icon $\boxed{x}$ on its top.
- An *integer cube* is of the form shown in Figure 5.8.b. It consists of an opaque green cube with an integer icon $\boxed{i}$ on its top.
- An *floatingpoint cube* is of the form shown in Figure 5.8.c. It consists of an opaque green cube with a floatingpoint icon $\boxed{f}$ on its top.



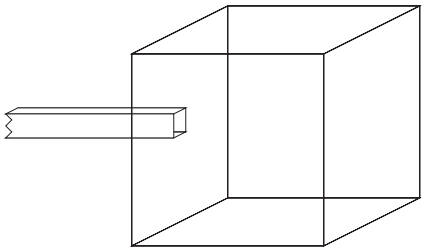
(a) Reference Cube



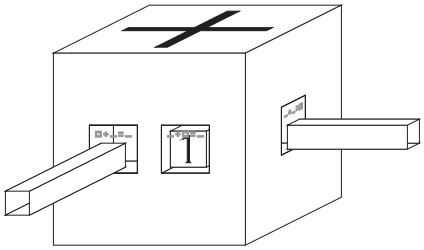
(b) Integer Cube



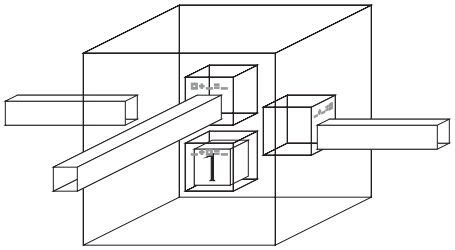
(c) Floatingpoint cube



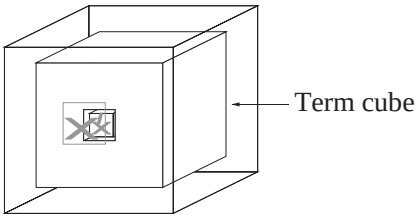
(d) Holder cube



(e) Application cube using reference cube



(f) Application cube using holder cube



(g) Port renaming cube

Figure 5.8: Term Cubes

- A *holder cube* is of the form shown in Figure 5.8.d. It consists of a transparent green cube which can either be empty or filled with another term cube, and can be connected to pipes. A port of a predicate definition cube (which is a special kind of holder) must always be empty [1].

- An *application cube* is a term cube with n ports ($n \geq 1$) set into its walls. The term cube must be either a reference cube (see Figure 5.8.e) or an empty holder cube (see Figure 5.8.f)[2]. The icon of each port is on the wall touching the term cube, which is transparent, the other walls are opaque.

- A *port renaming cube* is of the form shown in Figure 5.8.g. It consists of a term cube with (at least) ports named $\boxed{p_1}$, ..., $\boxed{p_n}$ ($n \geq 1$) in its walls (i.e. the term cube is an "application" cube, where no values are supplied to the ports $\boxed{p_1}$, ..., $\boxed{p_n}$), and a transparent cube surrounding it, with port icons $\boxed{p'_1}$, ..., $\boxed{p'_n}$ over $\boxed{p_1}$, ..., $\boxed{p_n}$.

A *pipe* is an opaque structure shaped like a spanning tree, having at least two ends, such that each end is connected to a holder cube.

A *conjunctive region* is a region of space containing a set of k definition cubes (see below) ($k \geq 0$), a set of m term cubes ($m \geq 0$), and a set of n pipes ($n \geq 0$).

A *plane*, as shown in Figure 5.9, is a conjunctive region surrounded by a transparent green box. A pipe may go through the wall of a plane, if it connects some cubes outside the plane with some cubes inside the plane.

A *Cube program*, as shown in Figure 5.10, is a conjunctive region.

A *predicate definition cube* has the form shown in Figure 5.11. It consists of a transparent green cube with icon $\boxed{P}$ and ports labeled $\boxed{p_1}$, ..., $\boxed{p_k}$ ($k \geq 0$), and m local definition cubes ($m \geq 0$) and n planes ($n \geq 0$) inside the predicate definition cube. Pipes can be used to connect the ports of the predicate definition cube to cubes inside the plane. However, pipes may not pass through the walls of the predicate definition cube, and they may not connect ports directly.

A *definition cube* is either a predicate definition cube or a type definition cube.

[1] We make this restriction only to ease the translation.

[2] It cannot be an integer or a floatingpoint cube, as these do not have a function type and thus cannot have ports. A filled holder cube can be replaced by the term cube inside it. An application cube can be merged with the outer application cube. A port renaming cube would be useless, as values can be supplied to ports regardless of their names.

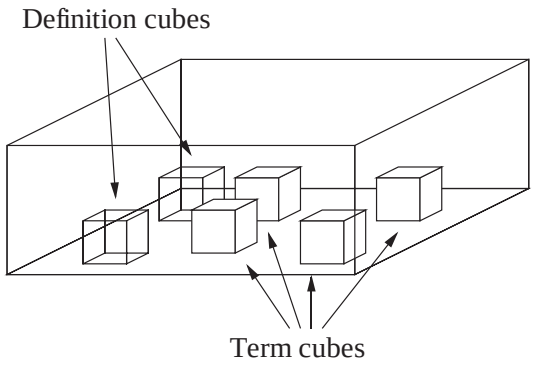


Figure 5.9: Visual Syntax of Planes

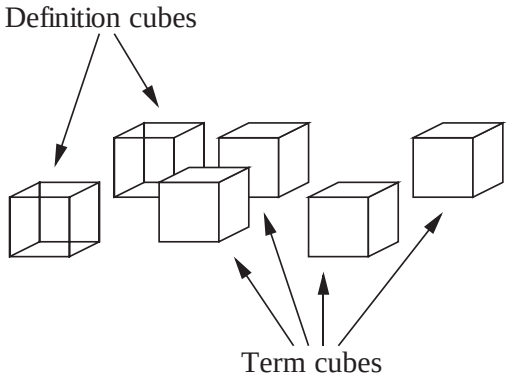


Figure 5.10: Program

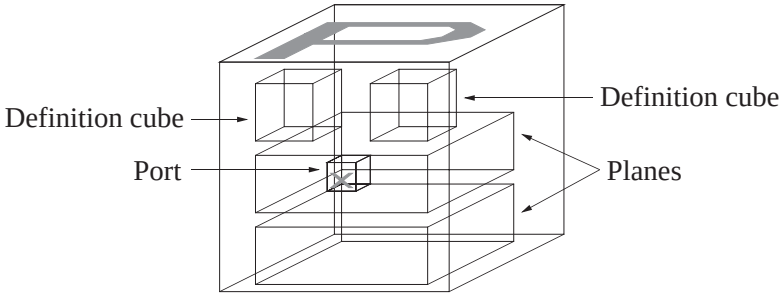


Figure 5.11: Visual Syntax of Predicate Definition Cubes

The translation method is as follows:

1. **(Term cubes)** Term cubes are translated with respect to an icon environment ρ, their translation may modify a set C of value constraints, and the result of the translation is a term.

   (a) **(Reference cubes)** A reference cube with icon $\boxed{x}$ translates to $\rho(\boxed{x})$.

   (b) **(Integer cubes)** An integer cube with icon $\boxed{i}$ translates to i.

   (c) **(Floatingpoint cubes)** A floatingpoint cube with icon $\boxed{f}$ translates to f.

   (d) **(Holder cubes)** Assume the holder cube is associated with the variable x. A top-level holder cube translates to *true* (a predefined predicate which always succeeds), a lower-level holder cube translates to x. If the holder cube is filled with a term cube which translates with respect to ρ and C to t, add the constraint $x = t$ to C.

   (e) **(Application cubes)** Assume the application cube consists of a term cube which translates with respect to ρ and C to t, and ports labeled $\boxed{p_1}, ..., \boxed{p_m}, \boxed{p_{m+1}}, ..., \boxed{p_n}$ $(0 \leq m \leq n, 1 \leq n)$, such that the holder cubes making up the ports $\boxed{p_1}, ..., \boxed{p_m}$ are either connected to a pipe, or filled with a term cube, that these holder cubes translate with respect to ρ and C into terms $t_1, ..., t_m$, and that the holder cubes making up the ports $\boxed{p_{m+1}}, ..., \boxed{p_n}$ are neither connected to a pipe, nor filled with a term cube[3]. Then the application cube translates to $t(p_1 = t_1)...(p_m = t_m)$.

   (f) **(Port renaming cubes)** A port renaming cube consisting of a term cube which translates with respect to ρ and C to t, ports labeled $\boxed{p_1}, ..., \boxed{p_n}$ set into the term cube, and icons $\boxed{p'_1}, ..., \boxed{p'_n}$ superimposing them, translates to $t\ (p_1 \rightarrow p'_1) \cdots (p_n \rightarrow p'_n)$.

2. **(Pipes)** Pipes are *resolved*, this process may modify a constraint set C. In order to resolve a pipe connecting n holder cubes associated with variables $x_1, ..., x_n$, add the constraints $x_1 = x_2, ..., x_1 = x_n$ to C.

[3]The translation scheme for application cubes reflects a design choice we had to make. In an alternative setting, each port in the picture, regardless whether it is supplied a value or not, is considered to be a binding of an actual to a formal parameter. In particular, a port which is empty, i.e. not supplied any value, is considered to be the binding of a variable with only one occurrence (like "_" in Prolog) to a formal parameter. The problem with this approach is that ports are visible in a Cube program only if and when they are used. This conflicts with our ideas about interactive editing of Cube programs: Unused ports in occurrences of (previously defined) predicates or constructors should be visible while editing, so that pipes can be connected to these ports.

3. **(Conjunctive regions)** Conjunctive regions are translated with respect to an icon environment ρ, the result of the translation is a conjunction. Assume a conjunctive region contains k definition cubes, m term cubes, and n pipes. Let $b_1, ..., b_p$ be the bindings yielded by the k definition cubes (see Definition 5.1.8). Let $\rho' = \rho[b_1, ..., b_p]$. Translate the k definition cubes with respect to ρ', obtaining definitions $D_1, ..., D_k$. Associate the holder cubes in the conjunctive region which are not part of the local definition cubes with new and distinct variables $x_1, ..., x_q$. Let C be an initially empty set of value constraints. Translate the m term cubes with respect to ρ' and C, obtaining terms $t_1, ..., t_m$, and resolve the n pipes with respect to C. Assume C is $\{x'_1 = t'_1, ..., x'_r = t'_r\}$ afterwards. Then the conjunctive region translates to

$$\textbf{letrec } D_1, ..., D_k \textbf{ in } \exists x_1, ..., x_q . t_1 \wedge ... \wedge t_m \wedge x'_1 = t'_1 \wedge ... \wedge x'_r = t'_r \wedge \mathit{true}$$

   We drop all the *true* in the conjunction, as long as at least one conjunct remains.

4. **(Planes)** Planes are translated with respect to an icon environment ρ. The translation of a plane is the same as the translation of its conjunctive region.

5. **(Programs)** Assume a program consists of a conjunctive region which translates with respect to the initial icon environment ρ_0 to c. Then the program translates to $\Leftarrow c$.

6. **(Definition cubes)** Definition cubes are translated with respect to an icon environment ρ, the result being a definition.

   (a) **(Type definition cubes)** The translation method for a type definition cube is described on page 79. This method actually ignores ρ.

   (b) **(Predicate definition cubes)** Assume a predicate definition cube with icon $\boxed{P}$, k ports labeled $\boxed{p_1}, ..., \boxed{p_k}$, m local definition cubes, and n planes. Associate new and distinct variables $x_1, ..., x_k$ with the ports. Let $b_1, ..., b_p$ be the bindings yielded by the m local definition cubes. Let $\rho' = \rho[\boxed{p_1} \mapsto x_1, ..., \boxed{p_k} \mapsto x_k, b_1, ..., b_p]$. Translate the m definition cubes with respect to ρ', obtaining definitions $D_1, ..., D_m$. Translate the n planes with respect to ρ', obtaining conjunctions $c_1, ..., c_n$. Then the predicate

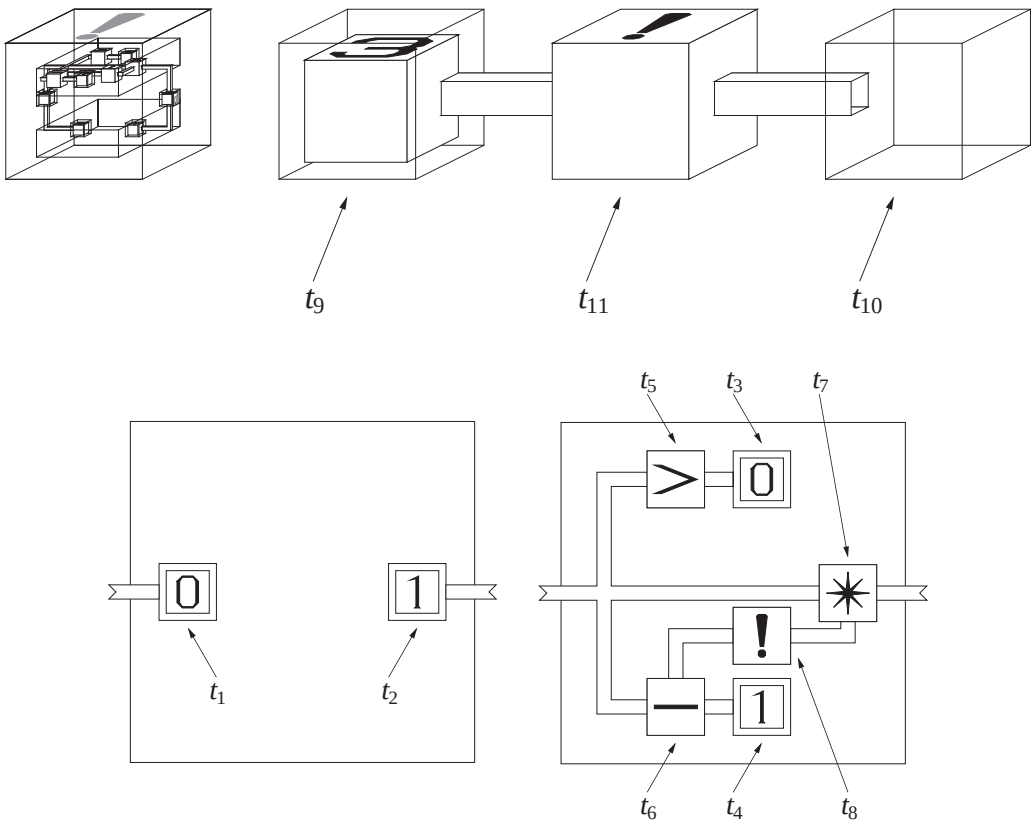


Figure 5.12: A Program Using the Factorial Predicate, and Details of It

definition cube translates to

$$\textbf{pred}\ P = \lambda\{p_1 = x_1, \cdots, p_k = x_k\}.\textbf{letrec}\ D_1, ..., D_m\ \textbf{in}\ c_1 \vee ... \vee c_n \vee \mathit{false}$$

false is a primitive predicate which always fails. If $n > 0$, we can drop *false*.

Example 5.1.2 Consider the factorial program shown in Figure 5.12. In order to translate it, we have (by rule 5) to translate the conjunctive region it is contained in with respect to the initial icon environment $\rho_0 = \{\boxed{\Leftrightarrow} \mapsto \mathit{minus}, \boxed{>} \mapsto \mathit{greater}, \boxed{*} \mapsto \mathit{times}, ...\}$ [4]. This conjunctive region contains one definition cube, which yields (by Definition 5.1.8) the binding

[4] For this example, no other parts of ρ_0 matter.

! $\mapsto$ *fact*. We set ρ_1 to be ρ_0[! $\mapsto$ *fact*] (by rule 3), and translate the definition cube with respect to ρ_1. This definition cube is a predicate definition cube with predicate icon !, ports labeled n and n!, and two planes, so rule 6b applies. We associate the variables x_1 and x_2 with the ports. As the predicate definition cube contains no local definition cubes, $\rho_2 = \rho_1$[n $\mapsto x_1$, n! $\mapsto x_2$]. Now we have to translate the two planes with respect to ρ_2. In order to translate the lower plane (see Figure 5.12), we translate (by rule 4) the conjunctive region it encloses. The conjunctive region contains no local definition cubes, two term cubes t_1 and t_2, and two pipes. So (by rule 3) ρ_2 does not have to be extended. We associate the two holder cubes t_1 and t_2 with variables x_3 and x_4, and initialize $C := \emptyset$. Next we translate t_1 with respect to ρ_2 and C. t_1 is a top-level holder cube, so it translates (by rule 1d) to *true*. It contains an integer cube with icon 1, which translates (by rule 1b) to 0, so we add (by rule 1d) the constraint $x_3 = 0$ to C. Similarly, t_2 translates to *true*, and $x_4 = 1$ is added to C. Now we have to resolve (by rule 3) the two pipes connected to t_1 and t_2. The first pipe connects to t_1 which is associated with x_3, and to the left port of the predicate definition cube, which is associated with x_1, so we add (by rule 2) the constraint $x_1 = x_3$ to C. Similarly, for the second pipe we add $x_2 = x_4$ to C. C is now $\{x_3 = 0, x_4 = 1, x_1 = x_3, x_2 = x_4\}$. So the conjunctive region translates (by rule 3) to

$$\exists x_3, x_4 . x_3 = 0 \wedge x_4 = 1 \wedge x_1 = x_3 \wedge x_2 = x_4$$

or e_1 for short. This is also (by rule 4) the translation of the lower plane of the predicate definition cube.

In order to translate the upper plane (see Figure 5.12), we have (by rule 4) again to translate with respect to ρ_2 the conjunctive region it encloses. The conjunctive region contains no local definition cubes, 6 term cubes $t_3, ..., t_8$, and 6 pipes. So (by rule 3) ρ_2 does not have to be extended. There are a total of 12 holder cubes in the region (two of them top-level holder cubes), and we associate the variables $x_5, ..., x_{16}$ with them. We initialize $C := \emptyset$. Then we translate the term cubes $t_3, ..., t_8$ with respect to ρ_2 and C. t_3 is a top-level holder cube, associated with x_5, and filled with an integer cube with icon 0, so it translates (by rules 1d and 1b) to *true*, and $x_5 = 0$ is added to C. t_4 is a top-level holder cube associated with x_6 and filled with an integer cube with icon 1, so it translates to *true*, and $x_6 = 1$ is added

to C. t_5 is an application cube, consisting of a reference cube with icon $\boxed{>}$, which translates (by rule 1a) with respect to ρ_2 and C to *greater*, and two ports labeled $\boxed{\rightarrow>}$ and $\boxed{>\leftarrow}$. Ports are lower-level holder cubes. The first port translates (by rule 1d) to x_7, the second to x_8. As both are empty, no constraints are added to C. But each of them is connected to a pipe, and thus really represents an application. So the application cube translates (by rule 1e) to $\mathit{greater}(\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8)$. Similarly, t_6 translates to $\mathit{minus}(\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11})$, t_7 translates to $\mathit{times}(\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14})$, and t_8 translates to $\mathit{fact}(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16})$. Note that the icon $\boxed{!}$ occurs recursively within the predicate definition cube defining it.

Next we have to translate the 6 pipes within the conjunctive region. The first is a pipe with four ends, connecting the holder cubes associated with the variables x_1 , x_7 , x_9 , and x_{12}, so we add the constraints $x_1 = x_7$, $x_1 = x_9$, and $x_1 = x_{12}$ to C. The 5 other pipes each have only two ends, connecting the holder cubes associated with the variables x_5 and x_8, x_6 and x_{10}, x_{11} and x_{15}, x_{13} and x_{16}, and x_{14} and x_2, respectively. Hence we add the constraints $x_5 = x_8$, $x_6 = x_{10}$, $x_{11} = x_{15}$, $x_{13} = x_{16}$, and $x_{14} = x_2$ to C. C is now $\{x_5 = 0, x_6 = 1, x_1 = x_7, x_1 = x_9, x_1 = x_{12}, x_5 = x_8, x_6 = x_{10}, x_{11} = x_{15}, x_{13} = x_{16}, x_{14} = x_2\}$. So (by rule 3) the conjunctive region translates to

$$
\begin{aligned}
&\exists x_5, ..., x_{16}.\\
&\qquad \mathit{greater}(\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8) \wedge\\
&\qquad \mathit{minus}(\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11}) \wedge\\
&\qquad \mathit{times}(\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14}) \wedge\\
&\qquad \mathit{fact}(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16}) \wedge\\
&\qquad x_5 = 0 \wedge x_6 = 1 \wedge x_1 = x_7 \wedge x_1 = x_9 \wedge x_1 = x_{12} \wedge\\
&\qquad x_5 = x_8 \wedge x_6 = x_{10} \wedge x_{11} = x_{15} \wedge x_{13} = x_{16} \wedge x_{14} = x_2
\end{aligned}
$$

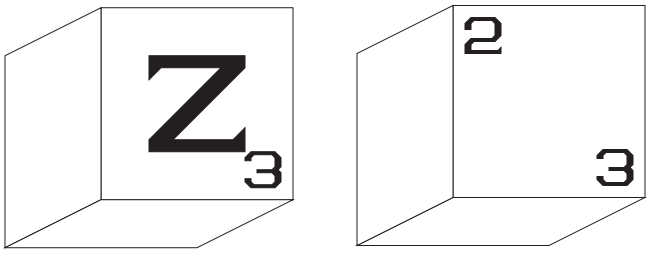
or e_2 for short. This is also (by rule 4) the translation of the upper plane. So the entire predicate definition cube translates (by rule 6b) to

$$\textbf{pred}\ \mathit{fact} = \lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}.e_1 \vee e_2$$

We are back to the translation of the overall conjunctive region. We associate (by rule 3) the four holder cubes in the conjunctive region with variables $x_{17}, ..., x_{20}$, and initialize $C := \emptyset$. Then we translate the term cubes t_9, t_{10}, and t_{11} with respect to ρ_1 and C. t_9 is a top-level holder cube, associated with x_{17}, and filled with an integer cube with icon `3`, which translates (by rule 1b) to 3. Hence t_9 translates (by rule 1d) to *true*, and $x_{17} = 3$ is added to C. t_{10} is an empty top-level holder cube, associated with x_{18}, and translates (by rule 1d) to *true*. t_{11} is an application cube, consisting of a reference cube with icon `!`, which translates (by rule 1a) to *fact*, and two ports labeled `n` and `n!`. These ports, both lower-level holder cubes, are associated with variables x_{19} and x_{20}, and thus translate (by rule 1d) to x_{19} and x_{20}. So t_{11} translates to $fact(\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20})$.

Now we need (again by rule 3) to resolve the two pipes. One of them connects the holder cube t_9, associated with x_{17}, to the left port of t_{11}, associated with x_{19}, so we add (by rule 2) $x_{17} = x_{19}$ to C. Similarly, for the other pipe, we add $x_{18} = x_{20}$ to C. C is now $\{x_{17} = 3, x_{17} = x_{19}, x_{18} = x_{20}\}$. So the outermost conjunctive region translates (by rule 3) to

$$\begin{array}{l}\textbf{letrec pred } fact = \lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}.e_1 \vee e_2 \\ \textbf{in } \exists x_{17}, ..., x_{20}. \\ \qquad fact(\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20}) \wedge x_{17} = 3 \wedge x_{17} = x_{19} \wedge x_{18} = x_{20}\end{array}$$



$x_3 : Int$ $x_3 : t_2$

Figure 5.13: Uninstantiated Variables

Hence the overall program translates (by rule 5) to

$$
\begin{array}{l}
\Leftarrow \textbf{letrec pred } \mathit{fact} = \\
\quad \lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}. \\
\quad\quad (\exists x_3, x_4. \\
\quad\quad\quad x_3 = 0 \wedge x_4 = 1 \wedge x_1 = x_3 \wedge x_2 = x_4) \vee \\
\quad\quad (\exists x_5, ..., x_{16}. \\
\quad\quad\quad \mathit{greater}(\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8) \wedge \\
\quad\quad\quad \mathit{minus}(\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11}) \wedge \\
\quad\quad\quad \mathit{times}(\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14}) \wedge \\
\quad\quad\quad \mathit{fact}(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16}) \wedge x_5 = 0 \wedge x_6 = 1 \wedge \\
\quad\quad\quad x_1 = x_7 \wedge x_1 = x_9 \wedge x_1 = x_{12} \wedge x_5 = x_8 \wedge x_6 = x_{10} \wedge x_{11} = x_{15} \wedge \\
\quad\quad\quad x_{13} = x_{16} \wedge x_{14} = x_2) \\
\textbf{in } \exists x_{17}, ..., x_{20}. \\
\quad \mathit{fact}(\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20}) \wedge x_{17} = 3 \wedge x_{17} = x_{19} \wedge x_{18} = x_{20}
\end{array}
$$

Section 5.3 describes an operational semantics which rewrites a textual program into a normal form, if there is one, and which obtains as a side effect the terms certain variables of the original program get replaced by when deriving this normal form. These terms can be integer and floating point constants, variables, and "closures". They shall then be displayed right inside the original Cube program. So, we need to translate the textual terms back into visual term cubes. This is done according to the following four rules:

1. An *integer constant* i is visualized by an integer term cube with icon $\boxed{i}$.

2. A *floatingpoint constant* f is visualized by a floatingpoint term cube with icon $\boxed{f}$.

3. A *variable* x_i of type τ is visualized by a type cube representing τ (see page 81), with the number i being in the lower right corner of the top side of the type cube. Figure 5.13 shows two such variable cubes.

4. A *"closure"* [5] results from applying either a constructor or a predicate to some arguments. The closure is represented by an application cube consisting of a reference cube referring to the constructor or predicate, and empty ports representing the remaining arguments.

5.2 Type Inference

This section describes a type inference system for Cube. It does this by giving a set of type inference rules for (a derivate of) L_0. The understanding is that a Cube program is well-typed if and only if its translation into L_0 by the translation method described in the previous section is well-typed.

First we want to establish some terminology. It turns out to be convenient if we collapse disjunctions, conjunctions, and terms of L_0 into one single syntactic category, namely *expressions*. Table 5.2 shows this simplified version of L_0, called L_1. Clearly, every word in L_0 is also a word in L_1 (but not vice versa).

Associated with every expression e is a *type scheme* σ; we denote this by $e : \sigma$ and call it a *typing*. A *type scheme* is of the form $\forall\alpha_1.\ \cdots\ \forall\alpha_n.\tau$ $(n \geq 0)$ and universally quantifies the type variables $\alpha_1, \cdots, \alpha_n$ in τ. A *type* can be a type variable α, a type constructor application $K\ \{\pi_1 = \tau_1, \cdots, \pi_n = \tau_n\}$, or a function type $\{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau$. The "base types" *Int*, *Float*, and *Prop* are treated as nullary type constructors.

[5] In the standard terminology, a closure is a value. However, in this context, we do not distinguish between terms and values.

E	$\in$	Program	
e	$\in$	Expression	
D	$\in$	Definition	
x	$\in$	Variable	
k	$\in$	Constructor $\subset$ Variable	
P	$\in$	Predicate $\subset$ Variable	
i	$\in$	Integer	
f	$\in$	Floatingpoint	
p	$\in$	Portname	
K	$\in$	Typeconstructor	
α	$\in$	Typevariable	
V	$\in$	Typevariant	
π	$\in$	Typeportname	
τ	$\in$	Type	
σ	$\in$	Typescheme	
E	::	$\Leftarrow e$	*Program*
e	::	i	*Integer constant*
	$\mid$	f	*Floating-point constant*
	$\mid$	x	*Variable*
	$\mid$	$e_1 = e_2$	*Unification*
	$\mid$	$e_1(p = e_2)$	*Application*
	$\mid$	$e\ (p \rightarrow p')$	*Parameter Renaming*
	$\mid$	$\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e$	*Abstraction*
	$\mid$	**letrec** $D_1, \cdots, D_n$ **in** e	*Definitions*
	$\mid$	$e_1 \vee \cdots \vee e_n$	*Disjunction*
	$\mid$	$e_1 \wedge \cdots \wedge e_n$	*Conjunction*
	$\mid$	$\exists x_1, \cdots, x_n.e$	*Existential quantification*
D	::	**pred** $P = e$	*Predicate definition*
	$\mid$	**type** $K\{\pi_1 = \alpha_1, \cdots, \pi_m = \alpha_m\} = V_1 + \cdots + V_n$	*Type definition*
V	::	$k\ \{p_1 : \tau_1, \cdots, p_n : \tau_n\}$	*Typevariant*
τ	::	α	*Type variable*
	$\mid$	$\{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau$	*Function type*
	$\mid$	$K\ \{\pi_1 = \tau_1, \cdots, \pi_n = \tau_n\}$	*Type constructor application*
σ	::	$\forall\alpha.\sigma$	*Type variable generalization*
	$\mid$	τ	*Type*

Table 5.2: Syntax of L_1

For example, the following typings hold:

$$1 : \mathit{Int}$$
$$3.14159 : \mathit{Float}$$
$$\mathit{fact} : \{\mathsf{fact}_1 : \mathit{Int}, \mathsf{fact}_2 : \mathit{Int}\} \rightarrow \mathit{Prop}$$

where *fact* refers to the factorial predicate translated in Example 5.1.2.

Now consider the expression $\lambda\{\mathsf{port}_1 = x_1\}.\mathit{true}$. This predicate can be applied to one argument, as in $(\lambda\{\mathsf{port}_1 = x_1\}.\mathit{true})(\mathsf{port}_1 = 5)$. But the type of the argument does not matter, the application will always be "well-typed". We call such a predicate *polymorphic*. The type scheme of this predicate is $\forall t_1.\{\mathsf{port}_1 : t_1\} \rightarrow \mathit{Prop}$. The universal quantifier indicates that t_1 can take on any type. The function type indicates that the predicate takes one argument of type t_1 at "port" port_1, and returns a truth-value.

Recall the list type definition from Example 5.1.1. It gives rise to two constructors, *cons* and *nil*. Their typings are

$$\mathit{cons} : \forall t_1.\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : \mathit{List}\{\mathsf{List}_1 = t_1\}\} \rightarrow \mathit{List}\{\mathsf{List}_1 = t_1\}$$
$$\mathit{nil} : \forall t_1.\mathit{List}\{\mathsf{List}_1 = t_1\}$$

These typings are derived directly from the definition of *List*, by universally quantifying the free type variables occurring in the parameter list of the type definition.

The expression $\mathit{cons}(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \mathit{nil})$ is well-typed: applying *cons* at "port" cons_1 to 5 specializes t_1 in the type of *cons* to *Int*, and removes the cons_1 argument from the function type, so that the resulting expression has type

$$\{\mathsf{cons}_2 : \mathit{List}\{\mathsf{List}_1 = \mathit{Int}\}\} \rightarrow \mathit{List}\{\mathsf{List}_1 = \mathit{Int}\}$$

Applying this expression at port cons_2 to *nil* works, as the type of *nil* can be specialized to be the same as the type of cons_2, so the resulting expression has type $\{\} \rightarrow \mathit{List}\{\mathsf{List}_1 = \mathit{Int}\}$ or simply $\mathit{List}\{\mathsf{List}_1 = \mathit{Int}\}$. But

$$\mathit{cons}(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \mathit{cons}(\mathsf{cons}_1 = 3.14159)(\mathsf{cons}_2 = \mathit{nil}))$$

is not well-typed, because t_1 cannot be specialized to *Int* and to *Float* at the same time.

The above example demonstrates a technique known as *currying*: "functions" (including predicates) which take multiple arguments can be applied to one argument at a time. This is reflected by the following law:

Definition 5.2.1 (Equivalence of function types)

1. $\{\} \to \tau$ and τ are equivalent
2. $\{p_1 : \tau_1, \cdots, p_m : \tau_m\} \to (\{p'_1 : \tau'_1, \cdots, p'_n : \tau'_n\} \to \tau)$ and
   $\{p_1 : \tau_1, \cdots, p_m : \tau_m, p'_1 : \tau'_1, \cdots, p'_n : \tau'_n\} \to \tau$ are equivalent

Unfortunately, the Hindley-Milner type inference system is more conservative than one would hope: it rules out some type-safe expressions, expressions which will not "go wrong" due to a type error. One particular restriction is that the variables bound by a set of mutually recursive definitions may occur only monomorphically in the bodies of the definitions. For example, in the expression

$$\textbf{letrec pred } foo = e_1, \textbf{pred } bar = e_2 \textbf{ in } e$$

all occurrences of *foo* and *bar* within e_1 and e_2 must have the same type. So

$$\begin{array}{l}\textbf{letrec pred } alwaysTrue = \lambda\{\mathsf{port}_1 = x_1\}.true, \\ \qquad \textbf{pred } trueAsWell = alwaysTrue(\mathsf{port}_1 = 5) \wedge alwaysTrue(\mathsf{port}_1 = 3.14159) \\ \textbf{in } trueAsWell\end{array}$$

is rejected by our type system as ill-typed, because *alwaysTrue* is used in *trueAsWell* once as a predicate over integers, and once as a predicate over floatingpoint numbers. The expression is, however, perfectly type-safe.

In order to minimize the impact of this restrictiveness, it is customary to split up each **letrec**, such that definitions which are not truly mutually recursive are not in the same **letrec**.

The above example could for instance be transformed into

$$\textbf{letrec pred } \mathit{alwaysTrue} = \lambda\{\mathsf{port}_1 = x_1\}.\mathit{true} \textbf{ in}$$
$$\textbf{letrec pred } \mathit{trueAsWell} = \mathit{alwaysTrue}(\mathsf{port}_1 = 5) \wedge \mathit{alwaysTrue}(\mathsf{port}_1 = 3.14159) \textbf{ in}$$
$$\mathit{trueAsWell}$$

which is well-typed in our type-system.

This transformation is accomplished by building a dependency graph for each **letrec**, such that each definition is represented by a vertex, and each use of a variable defined in the **letrec** is represented by an arc from the definition it is used in to its own definition. The strongly connected components of the graph are then identified and collapsed into single nodes. The resulting graph is acyclic, so it can be sorted topologically. The resulting ordering of nodes (each node now containing a set of definitions) is the desired ordering of **letrec**s, such that each **letrec** contains only truly mutually recursive definitions. Further details on this technique can be found in [64].

In the following, we assume that the expressions obtained from the picture-to-text translation have been transformed in such a way.

The type system we are describing now is formulated in form of type inference rules. This technique goes back to Damas and Milner [17]. We use the same notation as [22].

An *assumption* is an association of a variable x with a type scheme σ, denoted by $x : \sigma$ as described above. A denotes a finite set of unique assumptions. $A \vdash e : \sigma$ means "from A we can deduce that e has type scheme σ", and is called a sequent. $A.x : \sigma$ denotes the set of assumptions formed by removing any assumption about x from A, and then adding $x : \sigma$. $\frac{s_1 \cdots s_n}{s}$ is read "from the sequents s_1 and ... and s_n we can infer s". Finally, $[\tau/\alpha]\sigma$ is the result of substituting each free occurrence of the type variable α in the type scheme σ by the type τ.

Table 5.3 shows the type inference rules for L_1.

Example 5.2.1 Consider the program

$$\Leftarrow \quad \textbf{letrec pred } \mathit{alwaysTrue} = \lambda\{\mathsf{port}_1 = x_1\}.\mathit{true} \textbf{ in}$$
$$\mathit{alwaysTrue}(\mathsf{port}_1 = 5) \wedge \mathit{alwaysTrue}(\mathsf{port}_1 = 3.14159)$$

[INT] $A \vdash i : Int$

[FLOAT] $A \vdash f : Float$

[VAR] $A.x : \sigma \vdash x : \sigma$

[UNIF] $$\frac{A \vdash e_1 : \tau \quad A \vdash e_2 : \tau}{A \vdash (e_1 = e_2) : Prop}$$

[APP] $$\frac{A \vdash e_1 : \{p : \tau\} \rightarrow \tau' \quad A \vdash e_2 : \tau}{A \vdash (e_1(p = e_2)) : \tau'}$$

[DISJ] $$\frac{A \vdash e_1 : Prop \ \cdots \ A \vdash e_n : Prop}{A \vdash (e_1 \vee \cdots \vee e_n) : Prop}$$

[CONJ] $$\frac{A \vdash e_1 : Prop \ \cdots \ A \vdash e_n : Prop}{A \vdash (e_1 \wedge \cdots \wedge e_n) : Prop}$$

[EXIST] $$\frac{A \ . \ x_1 : \tau_1 \ . \ \cdots \ . \ x_n : \tau_n \vdash e : Prop}{A \vdash (\exists x_1, \cdots, x_n.e) : Prop}$$

[ABS] $$\frac{A \ . \ x_1 : \tau_1 \ . \ \cdots \ . \ x_n : \tau_n \vdash e : \tau}{A \vdash (\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e) : \{p_1 : \tau_1, \cdots, p_n : \tau_n\} \rightarrow \tau}$$

[PROG] $$\frac{A \vdash e : Prop}{A \vdash \Leftarrow e : Prop}$$

[SPEC] $$\frac{A \vdash e : \forall\alpha.\sigma}{A \vdash e : [\tau/\alpha]\sigma}$$

[TDEF] $$\frac{A \ . \ k_1 : (\forall\alpha_1, \cdots, \alpha_k.\{p_{11} : \tau_{11}, \cdots, p_{1n_1} : \tau_{1n_1}\} \rightarrow K \ \{\pi_1 = \alpha_1, \cdots, \pi_k = \alpha_k\}) \ . \ \cdots \vdash e : \tau}{A \vdash (\textbf{letrec type } K\{\pi_1 = \alpha_1, \cdots, \pi_k = \alpha_k\} = k_1 \ \{p_{11} : \tau_{11}, \cdots, p_{1n_1} : \tau_{1n_1}\} + \cdots \textbf{ in } e) : \tau}$$

[PDEF] $$\frac{\begin{array}{c} A \ . \ P_1 : \tau_1 \ . \ \cdots \ . \ P_n : \tau_n \vdash e_1 : \tau_1 \\ \vdots \\ A \ . \ P_1 : \tau_1 \ . \ \cdots \ . \ P_n : \tau_n \vdash e_n : \tau_n \\ A \ . \ P_1 : \forall\vec{\alpha_1}.\tau_1 \ . \ \cdots \ . \ P_n : \forall\vec{\alpha_n}.\tau_n \vdash e : \tau \end{array}}{A \vdash (\textbf{letrec pred } P_1 = e_1, \cdots, \textbf{pred } P_n = e_n \textbf{ in } e) : \tau}$$

where $\forall\vec{\alpha_i} \ = \ \forall\alpha_{i1}.\cdots.\alpha_{im_i}$ s.t. $\alpha_{i1}, \cdots, \alpha_{im_i}$ are not free in A

Table 5.3: Type Inference Rules for L_1

For brevity, we will abbreviate $\lambda\{\mathsf{port}_1 = x_1\}.true$ as e_1, $alwaysTrue(\mathsf{port}_1 = 5)$ as e_2, and $alwaysTrue(\mathsf{port}_1 = 3.14159)$ as e_3.

Our initial set of assumptions A_0 contains assumptions about the type schemes of the predefined predicates and constructors. In particular, as *true* and *false* are viewed as variables referring to predefined nullary predicates, A_0 will contain assumptions about them. For simplicity, in this example we assume that $A_0 = \{true : Prop\}$.

The overall expression is a program, so the [PROG] rule applies:

$$\frac{\textbf{letrec pred } alwaysTrue = e_1 \textbf{ in } e_2 \wedge e_3}{\{true : Prop\} \vdash (\Leftarrow \textbf{letrec pred } alwaysTrue = e_1 \textbf{ in } e_2 \wedge e_3) : Prop} \text{ [PROG]}$$

The next expression is a predicate definition, so the [PDEF] rule applies:

$$\frac{\{true : Prop, alwaysTrue : \tau_1\} \vdash e_1 : \tau_1 \qquad \{true : Prop, alwaysTrue : \forall\vec{\alpha}.\tau_1\} \vdash (e_2 \wedge e_3) : Prop}{\{true : Prop\} \vdash (\textbf{letrec pred } alwaysTrue = e_1 \textbf{ in } e_2 \wedge e_3) : Prop} \text{ [PDEF]}$$

e_1 is a λ-abstraction, so the [ABS] rule applies:

$$\frac{\{true : Prop, alwaysTrue : \tau_1, x_1 : \tau_2\} \vdash true : \tau_3}{\{true : Prop, alwaysTrue : \tau_1\} \vdash (\lambda\{\mathsf{port}_1 = x_1\}.true) : \{\mathsf{port}_1 : \tau_2\} \rightarrow \tau_3} \text{ [ABS]}$$

and τ_1 gets replaced by $\{\mathsf{port}_1 : \tau_2\} \rightarrow \tau_3$ throughout the proof. *true* is a variable, so the [VAR] rule applies:

$$\frac{}{\{true : Prop, alwaysTrue : \tau_1, x_1 : \tau_2\} \vdash true : Prop} \text{ [VAR]}$$

and τ_3 gets replaced by *Prop* throughout the proof. Now we get back to the body of the predicate definition. The set of assumptions $\{true : Prop, alwaysTrue : \forall\vec{\alpha}.\tau_1\}$ is by now $\{true : Prop, alwaysTrue : \forall\vec{\alpha}.\{\mathsf{port}_1 : \tau_2\} \rightarrow Prop\}$. Setting $\tau_2 = t_1$, we get

$$A_1 = \{true : Prop, alwaysTrue : \forall t_1.\{\mathsf{port}_1 : t_1\} \rightarrow Prop\}$$

The body of the **letrec** is a conjunction, so the [CONJ] rule applies:

$$\frac{A_1 \vdash e_2 : Prop \qquad A_1 \vdash e_3 : Prop}{A_1 \vdash (e_2 \wedge e_3) : Prop} \text{ [CONJ]}$$

e_2 is an application, so the [APP] rule applies:

$$\frac{A_1 \vdash \mathit{alwaysTrue} : \{\mathsf{port}_1 : \tau_4\} \to \mathit{Prop} \qquad A_1 \vdash 5 : \tau_4}{A_1 \vdash (\mathit{alwaysTrue}(\mathsf{port}_1 = 5)) : \mathit{Prop}} \text{ [APP]}$$

Next we apply the [SPEC] rule to the variable *alwaysTrue*:

$$\frac{A_1 \vdash \mathit{alwaysTrue} : \forall t_1.\{\mathsf{port}_1 : t_1\} \to \mathit{Prop}}{A_1 \vdash \mathit{alwaysTrue} : \{\mathsf{port}_1 : \mathit{Int}\} \to \mathit{Prop}} \text{ [SPEC]}$$

and we replace τ_4 by *Int* throughout the proof. Now we can apply the [VAR] rule:

$$\frac{}{\{\mathit{true} : \mathit{Prop}, \mathit{alwaysTrue} : \forall t_1.\{\mathsf{port}_1 : t_1\} \to \mathit{Prop}\} \vdash \mathit{alwaysTrue} : \forall t_1.\{\mathsf{port}_1 : t_1\} \to \mathit{Prop}} \text{ [VAR]}$$

For the argument 5 of the application, the [INT] rule applies:

$$\frac{}{A_1 \vdash 5 : \mathit{Int}} \text{ [INT]}$$

The type derivation of e_3 is similar, except that t_1 is specialized to floatingpoint numbers:

$$\frac{\dfrac{\dfrac{}{A_1 \vdash \mathit{alwaysTrue} : \forall t_1.\{\mathsf{port}_1 : t_1\} \to \mathit{Prop}}\text{ [VAR]}}{A_1 \vdash \mathit{alwaysTrue} : \{\mathsf{port}_1 : \mathit{Float}\} \to \mathit{Prop}}\text{ [SPEC]} \qquad \dfrac{}{A_1 \vdash 3.14159 : \mathit{Float}}\text{ [FLOAT]}}{A_1 \vdash (\mathit{alwaysTrue}(\mathsf{port}_1 = 3.14159)) : \mathit{Prop}}\text{ [APP]}$$

Example 5.2.2 For an example involving type definitions, consider the program

$$\Leftarrow \textbf{ letrec type } \mathit{List}\{\mathsf{List}_1 = t_1\} = \mathit{nil} + \mathit{cons}\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : \mathit{List}\{\mathsf{List}_1 = t_1\}\} \textbf{ in } \exists x_1.x_1 = \mathit{nil}$$

This program will succeed when evaluated, and by doing so, will instantiate x_1 to be *nil*.

As this example does not involve any predefined predicates or constructors, we assume for the sake of simplicity that the initial set of assumptions is empty.

If we use the following abbreviations:

$$
\begin{aligned}
A_1 = \{ \quad & cons : \forall t_1.\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : List\{\mathsf{List}_1 = t_1\}\} \to List\{\mathsf{List}_1 = t_1\}, \\
& nil : \forall t_1.List\{\mathsf{List}_1 = t_1\} \ \ \} \\
A_2 = \{ \quad & cons : \forall t_1.\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : List\{\mathsf{List}_1 = t_1\}\} \to List\{\mathsf{List}_1 = t_1\}, \\
& nil : \forall t_1.List\{\mathsf{List}_1 = t_1\}, x_1 : List\{\mathsf{List}_1 = t_1\} \ \ \}
\end{aligned}
$$

then the type derivation tree of the program looks as follows:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_2 \vdash x_1 : List\{\mathsf{List}_1 = t_1\}}\,[\text{VAR}] \qquad \dfrac{\dfrac{}{A_2 \vdash nil : \forall t_1.List\{\mathsf{List}_1 = t_1\}}\,[\text{VAR}]}{A_2 \vdash nil : List\{\mathsf{List}_1 = t_1\}}\,[\text{SPEC}]}{A_2 \vdash (x_1 = nil) : Prop}\,[\text{UNIF}]}{A_1 \vdash (\exists x_1.x_1 = nil) : Prop}\,[\text{EXIST}]}{\{\} \vdash (\textbf{letrec } \cdots \textbf{ in } \exists x_1.x_1 = nil) : Prop}\,[\text{TDEF}]}{\{\} \vdash (\Leftarrow \textbf{letrec } \cdots \textbf{ in } \exists x_1.x_1 = nil) : Prop}\,[\text{PROG}]
$$

5.3 Operational Semantics

This section formally describes the meaning the Cube programs, in form of an operational semantics for (a derivate of) L_0. The understanding is that the result of evaluating a Cube program can be determined by translating it into a textual form (by the rules provided in Section 5.1), evaluating this textual form, and then translating the result back into a picture (again as described in Section 5.1).

The operational semantics of Cube is given in form of a rewrite system with two rewrite rules: A rule for reducing terms, and a rule for resolving formulas. This rewrite system loosely orients itself on the operational semantics for Prolog given by Lloyd [46]. In Lloyd's operational semantics, whenever a variable gets "bound" to a term, all free occurrences of the variable are replaced by this term. This works fine in Prolog, as the term is guaranteed (by the "occurs check" built into unification) not to contain itself a free occurrence of the variable, and thus will not become infinite. In Cube, however, a problem arises: predicates are treated as terms. A predicate definition is viewed as a unification of a variable naming the predicate with the λ-abstraction defining it. So, defining a recursive predicate would be impossible, as it would mean unifying a variable with a term containing a free occurrence of this variable.

We resolve this problem by introducing a fixed-point operator into our textual language. The operator $\mathbf{fix}\, x\, .\, e$ reduces to e with each free occurrence of x in e replaced by $\mathbf{fix}\, x\, .\, e$, written as $e[\mathbf{fix}\, x\, .\, e/x]$. This operator allows us to transform a recursive predicate definition

$$foo = \cdots foo \cdots$$

into a non-recursive form

$$foo = \mathbf{fix}\, foo'\, .\, (\cdots foo' \cdots)$$

This technique is sufficient for dealing with directly recursive predicates. In order to deal with mutually recursive definitions of the form

$$\mathbf{letrec}\ x_1 = e_1, ..., x_n = e_n\ \mathbf{in}\ e$$

we form a tuple out of $e_1, ..., e_n$, fix this tuple against a new variable x, and replace each occurrence of x_i $(1 \leq i \leq n)$ in e_j $(1 \leq j \leq n)$ and e by $\mathbf{sel}\text{-}i\ x$, the selector function for the ith component of a tuple.

Recall that the introduction of the fixed-point operator was motivated by the idea that we want to replace variables by their values once they receive one. But we view *plus* etc. as variables referring to predefined predicates, i.e. bound variables. So, we need to replace them by a λ-abstraction, containing the predefined predicate itself. Similarly, we view a constructor name like *nil* which occurs within the scope of a type definition defining it as a variable referring to a λ-abstraction which contains a "primitive" constructor.

These issues give rise to an even more simplified version of our textual language L_0, called L_2. As in L_1, disjunctions, conjunction, and terms are collapsed into one syntactic category, namely expressions. In addition, L_2 eliminates letrecs, types, and type definitions. On the flip side, it introduces a fixed-point operator, tuple constructors and selectors, and primitive predicates and constructors. Table 5.4 gives the syntax of L_2.

Programs are translated from L_0 to L_2 as follows:

Definition 5.3.1 (Translation from L_0 to L_2) The left-associative *constructor expansion operator* $\oplus$: Expression $\rightarrow$ Variant $\rightarrow$ Expression is defined as follows:

E	$\in$	Program	
e	$\in$	Expression	
x	$\in$	Variable	
i	$\in$	Integer	
f	$\in$	Floatingpoint	
p	$\in$	Portname	
$\overline{k}$	$\in$	Prim-Constructor	
$\overline{P}$	$\in$	Prim-Predicate	

E	::	$\Leftarrow e$	*Program*
e	::	i	*Integer constant*
	$\mid$	f	*Floatingpoint constant*
	$\mid$	x	*Variable*
	$\mid$	$e_1 = e_2$	*Unification*
	$\mid$	$e_1(p = e_2)$	*Application*
	$\mid$	$e\ (p \rightarrow p')$	*Parameter renaming*
	$\mid$	$\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e$	*Abstraction*
	$\mid$	$e_1 \vee ... \vee e_n$	*Disjunction*
	$\mid$	$e_1 \wedge ... \wedge e_n$	*Conjunction*
	$\mid$	$\exists x_1, ..., x_n.e$	*Existential quantification*
	$\mid$	$\overline{k}\ e_1 ... e_n$	*Prim. constructor application*
	$\mid$	$\overline{P}\ e_1 ... e_n$	*Prim. predicate application*
	$\mid$	**fix** $x\ .\ e$	*Fixed-point operator application*
	$\mid$	n-**tuple** $e_1 ... e_n$	*Tuple construction*
	$\mid$	**sel-**i e	*Tuple destruction*

Table 5.4: Syntax of L_2

$$e \ \oplus \ (k \ \{p_1 : \tau_1, \cdots, p_n : \tau_n\}) \ \equiv \ e[\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.\overline{k} \ x_1 \ ... \ x_n \ / \ k]$$

where $\overline{k}$ is a new primitive constructor and $x_1, ..., x_n$ are new and distinct variables.

The left-associative *type-definition expansion operator* $\odot$: Expression $\rightarrow$ Definition $\rightarrow$ Expression is defined as follows:

$$e \ \odot \ (\textbf{type} \ K\{\pi_1 = \alpha_1, \cdots, \pi_m = \alpha_m\} = V_1 + \cdots + V_n) \ \equiv \ e \oplus V_1 \oplus ... \oplus V_n$$

The *letrec expansion function expand* is defined as follows: given an expression e_0, replace each subexpression e of the form **letrec** $D_1^T, ..., D_m^T, ..., D_1^P, ..., D_n^P$ **in** e', where $D_1^T, ..., D_m^T$ are type definitions, and $D_1^P, ..., D_n^P$ are predicate definitions, such that each D_i^P is of the form **pred** $P_i = e_i$, by

$$(e' \odot D_1^T \odot ... \odot D_m^T)[\textbf{sel-}1 \ \mathcal{F}/P_1, ..., \textbf{sel-}n \ \mathcal{F}/P_n]$$

where $\mathcal{F} = \textbf{fix} \, x \, .(n\textbf{-tuple} \ e'_1 \ ... \ e'_n)$, $\quad e'_i = (e_i \odot D_1^T \odot ... \odot D_m^T)[\textbf{sel-}1 \ x \ /P_1, ..., \textbf{sel-}n \ x \ /P_n]$ $(1 \leq i \leq n)$, and x is a new variable.

Given a program $E \in L_0$ of the form $\Leftarrow c$, let

$$c' = c[\lambda\{\mathsf{plus}_1 = x'_1, \mathsf{plus}_2 = x'_2, \mathsf{plus}_3 = x'_3\}.\overline{plus} \ x'_1 \ x'_2 \ x'_3 \ / \ plus, ...]$$

where $x'_1, \cdots$ are new and distinct variables, then $\Leftarrow expand(c')$ is the *translation of* E *to* L_2.

Example 5.3.1 Example 5.1.2 showed the translation of the Cube program shown in Figure 5.12, and representing a definition of factorial and the query "factorial of 3" into a textual

form, namely

$$
\begin{array}{l}
\Leftarrow \textbf{letrec pred } \mathit{fact} = \\
\quad \lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}. \\
\qquad (\exists x_3, x_4 . x_3 = 0 \wedge x_4 = 1 \wedge x_1 = x_3 \wedge x_2 = x_4) \vee \\
\qquad (\exists x_5, ..., x_{16}. \\
\qquad\quad \mathit{greater}(\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8) \wedge \\
\qquad\quad \mathit{minus}(\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11}) \wedge \\
\qquad\quad \mathit{times}(\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14}) \wedge \\
\qquad\quad \mathit{fact}(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16}) \wedge \\
\qquad\quad x_5 = 0 \wedge x_6 = 1 \wedge x_1 = x_7 \wedge x_1 = x_9 \wedge x_1 = x_{12} \wedge x_5 = x_8 \wedge \\
\qquad\quad x_6 = x_{10} \wedge x_{11} = x_{15} \wedge x_{13} = x_{16} \wedge x_{14} = x_2) \\
\textbf{in } \exists x_{17}, ..., x_{20}. \mathit{fact}(\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20}) \wedge x_{17} = 3 \wedge x_{17} = x_{19} \wedge x_{18} = x_{20}
\end{array}
$$

Replacing the variables referring to predefined predicates by the appropriate λ-abstractions yields

$$
\begin{array}{l}
\Leftarrow \textbf{letrec pred } \mathit{fact} = \\
\quad \lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}. \\
\qquad (\exists x_3, x_4 . x_3 = 0 \wedge x_4 = 1 \wedge x_1 = x_3 \wedge x_2 = x_4) \vee \\
\qquad (\exists x_5, ..., x_{16}. \\
\qquad\quad (\lambda\{\mathsf{greater}_1 = x_1, \mathsf{greater}_2 = x_2\}.\overline{\mathit{greater}}\ x_1\ x_2) \\
\qquad\qquad (\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8) \wedge \\
\qquad\quad (\lambda\{\mathsf{minus}_1 = x_1, \mathsf{minus}_2 = x_2, \mathsf{minus}_3 = x_3\}.\overline{\mathit{minus}}\ x_1\ x_2\ x_3) \\
\qquad\qquad (\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11}) \wedge \\
\qquad\quad (\lambda\{\mathsf{times}_1 = x_1, \mathsf{times}_2 = x_2, \mathsf{times}_3 = x_3\}.\overline{\mathit{times}}\ x_1\ x_2\ x_3) \\
\qquad\qquad (\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14}) \wedge \\
\qquad\quad \mathit{fact}(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16}) \wedge \\
\qquad\quad x_5 = 0 \wedge x_6 = 1 \wedge x_1 = x_7 \wedge x_1 = x_9 \wedge x_1 = x_{12} \wedge x_5 = x_8 \wedge \\
\qquad\quad x_6 = x_{10} \wedge x_{11} = x_{15} \wedge x_{13} = x_{16} \wedge x_{14} = x_2) \\
\textbf{in } \exists x_{17}, ..., x_{20}. \mathit{fact}(\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20}) \wedge x_{17} = 3 \wedge x_{17} = x_{19} \wedge x_{18} = x_{20}
\end{array}
$$

Removing recursive definitions then yields a program in L_2, namely

$$
\begin{array}{l}
\Leftarrow \exists x_{17}, ..., x_{20}. \\
\quad (\textbf{sel-}1\ (\textbf{fix}\, x_0\, .\, (1\textbf{-tuple}\ (\lambda\{\mathsf{fact}_1 = x_1, \mathsf{fact}_2 = x_2\}. \\
\qquad\qquad (\exists x_3, x_4 . x_3 = 0 \wedge x_4 = 1 \wedge x_1 = x_3 \wedge x_2 = x_4) \vee \\
\qquad\qquad (\exists x_5, ..., x_{16}. \\
\qquad\qquad\quad (\lambda\{\mathsf{greater}_1 = x_1, \mathsf{greater}_2 = x_2\}.\overline{greater}\ x_1\ x_2) \\
\qquad\qquad\qquad (\mathsf{greater}_1 = x_7)(\mathsf{greater}_2 = x_8) \wedge \\
\qquad\qquad\quad (\lambda\{\mathsf{minus}_1 = x_1, \mathsf{minus}_2 = x_2, \mathsf{minus}_3 = x_3\}.\overline{minus}\ x_1\ x_2\ x_3) \\
\qquad\qquad\qquad (\mathsf{minus}_1 = x_9)(\mathsf{minus}_2 = x_{10})(\mathsf{minus}_3 = x_{11}) \wedge \\
\qquad\qquad\quad (\lambda\{\mathsf{times}_1 = x_1, \mathsf{times}_2 = x_2, \mathsf{times}_3 = x_3\}.\overline{times}\ x_1\ x_2\ x_3) \\
\qquad\qquad\qquad (\mathsf{times}_1 = x_{12})(\mathsf{times}_2 = x_{13})(\mathsf{times}_3 = x_{14}) \wedge \\
\qquad\qquad\quad (\textbf{sel-}1\ x_0)(\mathsf{fact}_1 = x_{15})(\mathsf{fact}_2 = x_{16}) \wedge \\
\qquad\qquad\quad x_5 = 0 \wedge x_6 = 1 \wedge x_1 = x_7 \wedge x_1 = x_9 \wedge x_1 = x_{12} \wedge x_5 = x_8 \wedge \\
\qquad\qquad\quad x_6 = x_{10} \wedge x_{11} = x_{15} \wedge x_{13} = x_{16} \wedge x_{14} = x_2)))))) \\
\qquad\quad (\mathsf{fact}_1 = x_{19})(\mathsf{fact}_2 = x_{20}) \wedge \\
\quad x_{17} = 3 \wedge x_{17} = x_{19} \wedge x_{18} = x_{20}
\end{array}
$$

The next issue we have to deal with is the reduction of terms. In Prolog, this issue does not arise, as terms are uninterpreted. In Cube, however, terms can be applications of λ-abstractions to arguments, so we need to reduce them to a normal form. This process is essentially what is known as β-reduction in the λ-calculus. The following three definitions formalize the reduction concept:

Definition 5.3.2 The *substitution of e for x in e'*, written $e'[e/x]$, is defined as follows:

$$
\begin{array}{rcl}
i[e/x] & = & i \\
f[e/x] & = & f \\
x[e/x] & = & e \\
x'[e/x] & = & x' \quad \text{if } x \neq x' \\
(e_1 = e_2)[e/x] & = & (e_1[e/x]) = (e_2[e/x]) \\
(e_1(p = e_2))[e/x] & = & (e_1[e/x])(p = (e_2[e/x])) \\
(e'\ (p \rightarrow p'))[e/x] & = & (e'[e/x])\ (p \rightarrow p') \\
(e_1 \vee ... \vee e_n)[e/x] & = & ((e_1[e/x]) \vee ... \vee (e_n[e/x])) \\
(e_1 \wedge ... \wedge e_n)[e/x] & = & ((e_1[e/x]) \wedge ... \wedge (e_n[e/x])) \\
(\overline{k}\ e_1 ... e_n)[e/x] & = & (\overline{k}\ (e_1[e/x]) ... (e_n[e/x])) \\
(\overline{P}\ e_1 ... e_n)[e/x] & = & (\overline{P}\ (e_1[e/x]) ... (e_n[e/x])) \\
(n\textbf{-tuple}\ e_1 ... e_n)[e/x] & = & (n\textbf{-tuple}\ (e_1[e/x]) ... (e_n[e/x])) \\
(\textbf{sel-}i\ e')[e/x] & = & (\textbf{sel-}i\ e'[e/x]) \\
\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e'[e/x] & = & \lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e' \\
 & & \text{if } x = x_i \text{ for any } 1 \leq i \leq n \\
\lambda\{p_1 = x_1, \cdots, p_n = x_n\}.e'[e/x] & = & \lambda\{p_1 = x'_1, \cdots, p_n = x'_n\}.(e'[x'_1/x_1][x'_n/x_n][e/x]) \\
 & & \text{if for all } 1 \leq i \leq n,\ x \neq x_i \text{ and } x'_i \text{ is not free in } e \text{ or } e' \\
\exists x_1, ..., x_n.e'[e/x] & = & \exists x_1, ..., x_n.e' \\
 & & \text{if } x = x_i \text{ for any } 1 \leq i \leq n \\
\exists x_1, ..., x_n.e'[e/x] & = & \exists x'_1, ..., x'_n.(e'[x'_1/x_1][x'_n/x_n][e/x]) \\
 & & \text{if for all } 1 \leq i \leq n,\ x \neq x_i \text{ and } x'_i \text{ is not free in } e \text{ or } e' \\
\textbf{fix}\, x\, .e'[e/x] & = & \textbf{fix}\, x\, .e' \\
\textbf{fix}\, x'\, .e'[e/x] & = & \textbf{fix}\, x''\, .(e'[x''/x'][e/x]) \\
 & & \text{if } x \neq x' \text{ and } x'' \text{ is not free in } e \text{ or } e'
\end{array}
$$

Definition 5.3.3 The *reduction-rule* $\stackrel{\text{red}}{\Leftrightarrow}$ denotes a relation between two expressions, and is defined as follows:

$[APP]$ $(\lambda\{p_1 = x_1, p_2 = x_2, ..., p_i = x_i\}.e)(p_1 = e') \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow} \lambda\{p_2 = x_2, ..., p_i = x_i\}.e[e'/x_1]$

$[CAST]$ $(\lambda\{p_1 = x_1, p_2 = x_2, ..., p_i = x_i\}.e)\ (p_1 \rightarrow p_1') \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow} (\lambda\{p_1' = x_1, p_2 = x_2, ..., p_i = x_i\}.e)$

$[SEL]$ $(\textbf{sel-}i\ (n\textbf{-tuple}\ \ e_1 \cdots e_i \cdots e_n)) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow} e_i$

$[FIX]$ $(\textbf{fix}\, x\, .\, e) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow} e[\textbf{fix}\, x\, .\, e/x]$

We write $e \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow} e'$ if a subexpression of e is reduced to create e'. $\overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$ denotes the compatible closure of $\overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$.

Definition 5.3.4 An expression e is said to be in *normal form* if none of its subexpressions is an application or a parameter renaming.

Example 5.3.2 The following two expressions both reduce into normal forms:

1. $(\lambda\{\mathsf{arg} = x_1\}.x_1 = 5)\ (\mathsf{arg} \rightarrow \mathsf{newarg}) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$
   $(\lambda\{\mathsf{newarg} = x_1\}.x_1 = 5)$

2. $(\lambda\{\mathsf{cons}_1 = x_1, \mathsf{cons}_2 = x_2\}.\overline{cons}\ x_1\ x_2)(\mathsf{cons}_1 = 3)(\mathsf{cons}_2 =$
   $(\lambda\{\mathsf{cons}_1 = x_1, \mathsf{cons}_2 = x_2\}.\overline{cons}\ x_1\ x_2)(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \overline{nil})) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$
   $(\lambda\{\mathsf{cons}_2 = x_2\}.\overline{cons}\ 3\ x_2)(\mathsf{cons}_2 =$
   $(\lambda\{\mathsf{cons}_1 = x_1, \mathsf{cons}_2 = x_2\}.\overline{cons}\ x_1\ x_2)(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \overline{nil})) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$
   $(\overline{cons}\ 3\ (\lambda\{\mathsf{cons}_1 = x_1, \mathsf{cons}_2 = x_2\}.\overline{cons}\ x_1\ x_2)(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \overline{nil})) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$
   $(\overline{cons}\ 3\ (\lambda\{\mathsf{cons}_2 = x_2\}.\overline{cons}\ 5\ x_2)(\mathsf{cons}_2 = \overline{nil})) \overset{\text{red}}{\Leftrightarrow\!\!\rightarrow}$
   $(\overline{cons}\ 3\ (\overline{cons}\ 5\ \overline{nil}))$

The **fix** operator is used only to express recursive predicates nonrecursively. Such a predicate is conceptually an infinite structure, but no real "computation" has to be performed to build this structure.

Note that not every expression reduces to a normal form: $x_1(\mathsf{arg} = 5)$ does not reduce to a normal form, as it is an application which cannot be rewritten, because the $[APP]$ rule does not apply. Once x_1 gets replaced by a value, reduction can continue, and the expression might rewrite into a normal form.

Unification in Cube is slightly more complicated than unification in Prolog: two expressions unify if they can be reduced to normal forms which unify in the conventional sense. For example,

$(\lambda\{\mathsf{cons}_1 = x_1, \mathsf{cons}_2 = x_2\}.\overline{cons}\ x_1\ x_2)(\mathsf{cons}_1 = 5)(\mathsf{cons}_2 = \overline{nil})$ and $(\overline{cons}\ 5\ \overline{nil})$ unify. But as not every expression can be reduced into a normal form, it is not always possible to decide whether two expressions unify. For instance, $x_3(\mathsf{cons}_1 = 5)$ and $x_4(\mathsf{cons}_2 = \overline{nil})$ unify, if x_3 gets bound to $(\lambda\{\mathsf{cons}_1 = x_1\}.\overline{cons}\ x_1\ \overline{nil})$ and x_4 gets bound to $(\lambda\{\mathsf{cons}_2 = x_2\}.\overline{cons}\ 5\ x_2)$.

So, unifying two expressions might succeed, fail, or be simply undecidable for the time being. This is made precise in the following definitions:

Definition 5.3.5 A *substitution* $\theta = \{x_1 \mapsto e_1, ..., x_n \mapsto e_n\}$ is a mapping from variables to expressions such that $x_1, ..., x_n$ are distinct and do not occur free in $e_1, ..., e_n$.

Definition 5.3.6 Let $\theta = \{x_1 \mapsto e_1, ..., x_n \mapsto e_n\}$ be a substitution and e be an expression. Then $e\theta$, the *instance of* e *by* θ, is the expression obtained from e by simultaneously replacing each free occurrence of the variable x_i in e by the expression e_i $(i = 1, ..., n)$.

Definition 5.3.7 Let θ and θ' be two substitutions, Then $\theta \circ \theta'$, the *composition* of θ and θ', is defined such that for every expression e, $e(\theta \circ \theta') = (e\theta')\theta$.

Definition 5.3.8 (Unification) Consider two expressions e_1 and e_2:

- e_1 and e_2 *unify with a most general unifier* θ if e_1 reduces to a normal form e_1', e_2 reduces to a normal form e_2', and there is a substitution θ such that $e_1'\theta = e_2'\theta$, and for every θ' such that $e_1'\theta' = e_2'\theta'$, there is a θ'' such that $\theta' = \theta'' \circ \theta$.
- e_1 and e_2 *potentially unify* if there is a substitution θ such that $e_1\theta \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} e$ and $e_2\theta \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} e$.
- e_1 and e_2 *fail to unify* if they neither unify nor potentially unify.

Example 5.3.3

1. $(\lambda\{\mathsf{cons}_1 = x_1\}.\overline{cons}\ x_1\ \overline{nil})(\mathsf{cons}_1 = x_3)$ and $(\overline{cons}\ x_4\ x_5)$ unify, as $(\lambda\{\mathsf{cons}_1 = x_1\}.\overline{cons}\ x_1\ \overline{nil})(\mathsf{cons}_1 = x_3) \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (\overline{cons}\ x_3\ \overline{nil})$, which is in normal form, $(\overline{cons}\ x_4\ x_5)$ is in normal form, and the two normal forms unify with most general unifier $\theta = \{x_3 \mapsto x_4, x_5 \mapsto \overline{nil}\}$.
2. $x_3(\mathsf{cons}_1 = 5)$ and x_4 potentially unify, as under the substitution $\theta = \{x_3 \mapsto (\lambda\{\mathsf{cons}_1 = x_1\}.\overline{cons}\ x_1\overline{nil}), x_4 \mapsto (\overline{cons}\ 5\ \overline{nil})\}$, they both reduce to the same normal form.

3. $x_2(\mathsf{arg} = 5)$ and $x_2(\mathsf{arg} = 3)$ potentially unify, as under the substitution $\theta = \{x_2 \mapsto (\lambda\{\mathsf{arg} = x_1\}.true)\}$, they both reduce to the same normal form.

4. $(\overline{cons}\ 1\ \overline{nil})$ and $\overline{nil}$ fail to unify.

5. $(\lambda\{\mathsf{cons}_2 = x_2\}.\overline{cons}\ 1\ \overline{nil})$ and $(\lambda\{\mathsf{cons}_2 = x_2\}.\overline{cons}\ 2\ \overline{nil})$ fail to unify.

The second rewrite rule used in Cube's operational semantics is the resolution rule. This rule is again loosely based on Lloyd [46]. Our formalism, however, differs in a number of aspects. One of them is the way the "state" of a rewriting is represented. While Lloyd simply uses a set of atomic formulas to represent the goals, we use a set of expressions to represent the goals, together with a tuple of expressions, which reflects the bindings performed on variables of the initial query. These bindings are needed for visualizing the outcome of a computation. Formally, this "state" is defined as follows:

Definition 5.3.9 A *goal* g is a truth-valued expression. A *configuration* C is either a pair $(\{g_1, ..., g_m\}, \langle e_1, ..., e_n\rangle)$ consisting of a set of goals and a sequence of expressions, or the *failure-configuration* **failure**. **failure** denotes a failed proof., $(\{g_1, ..., g_m\}, \langle e_1, ..., e_n\rangle)$ informally means that the goals $g_1, ..., g_m$ still have to be resolved in order to complete the proof at hand, and the free variables $x_1, ..., x_n$ in the formula to be proven have so far received values $e_1, ..., e_n$.

Given a program $E \in L_2$ of the form $\Leftarrow \exists x_1, ..., x_m.e_1 \wedge ... \wedge e_n$ $(m \geq 0, n \geq 1)$ the *initial configuration* C_E of E is of the form $(\{e_1, ..., e_n\}, \langle x_1, ..., x_m\rangle)$.

Before being able to formalize the resolution rule, we need a few more definitions:

Definition 5.3.10 An expression is said to be *ground* if it does not contain any free variables.

Definition 5.3.11 The *many-step-resolution-rule* $\overset{\text{res}}{\Leftrightarrow\!\!\!\!\Rightarrow}$ denotes a relation between two configurations, and is defined in terms of the one-step-resolution-rule $\overset{\text{res}}{\Leftrightarrow}$, which will be defined later, as follows:

$$C_0 \overset{\text{res}}{\Leftrightarrow\!\!\!\!\Rightarrow} C_n \Leftrightarrow \exists C_1, ..., C_{n-1}.C_0 \overset{\text{res}}{\Leftrightarrow} C_1 \overset{\text{res}}{\Leftrightarrow} ... \overset{\text{res}}{\Leftrightarrow} C_{n-1} \overset{\text{res}}{\Leftrightarrow} C_n \wedge \neg\exists C_{n+1}.C_n \overset{\text{res}}{\Leftrightarrow} C_{n+1}$$

Definition 5.3.12 The *solution set* $\mathrm{Sol}(C)$ of a configuration C is defined as follows:

$$\mathrm{Sol}(C) = \{\langle e_1, ..., e_n\rangle | C \overset{\text{res}}{\Leftrightarrow\!\!\!\!\Rightarrow} (\{\}, \langle e_1, ..., e_n\rangle)\}$$

The *deadlock set* $\mathrm{Dead}(C)$ of a configuration C is defined as follows:

$$\mathrm{Dead}(C) \;=\; \{(\{g_1,...,g_m\},\langle e_1,...,e_n\rangle) | C \;\overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow}\; (\{g_1,...,g_m\},\langle e_1,...,e_n\rangle) \;\wedge\; m>0\}$$

Definition 5.3.13 A configuration C *fails* if $\mathrm{Sol}(C)=\emptyset$, $\mathrm{Dead}(C)=\emptyset$, and there is no infinite sequence $C \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} C' \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} C'' \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} \cdots$. A configuration C *succeeds* if $\mathrm{Sol}(C)\neq\emptyset$.

Finally, we can formulate the resolution rule:

Definition 5.3.14 The *one-step-resolution-rule* $\overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow}$ relates two configurations and is defined as follows:

1. (Disjunction)
   $(\{g_1,\cdots,g_i,\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle) \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} (\{g_1,\cdots,g_{i-1},e_j,g_{i+1},\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle)$ [6]
   if $g_i \overset{\mathrm{red}}{\Leftrightarrow\!\!\!\!\rightarrow} e_1 \vee ... \vee e_j \vee ... \vee e_n \;(1\leq j\leq n)$ [7]

2. (Conjunction)
   $(\{g_1,\cdots,g_i,\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle) \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} (\{g_1,\cdots,g_{i-1},e_1,...,e_t,g_{i+1},\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle)$
   if $g_i \overset{\mathrm{red}}{\Leftrightarrow\!\!\!\!\rightarrow} e_1 \wedge ... \wedge e_t$

3. (Existential Quantification)
   $(\{g_1,\cdots,g_i,\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle) \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} (\{g_1,\cdots,g_{i-1},e',g_{i+1},\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle)$
   if $g_i \overset{\mathrm{red}}{\Leftrightarrow\!\!\!\!\rightarrow} \exists x_1,...,x_n.e$
   where $e'=e[x'_1/x_1,...,x'_n/x_n]$ and $x'_1,...,x'_n$ are new and distinct variables.

4. (Unification)

   (a) $(\{g_1,\cdots,g_i,\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle) \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow} (\{g_1\theta,\cdots,g_{i-1}\theta,g_{i+1}\theta,\cdots,g_a\theta\},\langle\hat{e_1}\theta,\cdots,\hat{e_b}\theta\rangle)$
   if $g_i \overset{\mathrm{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (e_1=e_2)$ and e_1 and e_2 unify with a most general unifier θ.

   (b) $(\{g_1,\cdots,g_i,\cdots,g_a\},\langle\hat{e_1},\cdots,\hat{e_b}\rangle) \overset{\mathrm{res}}{\Leftrightarrow\!\!\!\!\rightarrow}$ **failure**
   if $g_i \overset{\mathrm{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (e_1=e_2)$ and e_1 and e_2 fail to unify.

5. (Primitive Negation)

[6] i is chosen arbitrarily in all the $\overset{\mathrm{res}}{\Longrightarrow}$ rules, leading to one degree of nondeterminism.
[7] j is chosen arbitrarily in this rule, leading to a second degree of nondeterminism.

(a) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1, \cdots, g_{i-1}, g_{i+1}, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{not}\ e)$ and e is ground and the configuration $C' = (\{e\}, \langle\rangle)$ fails.

(b) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow}$ **failure**
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{not}\ e)$ and e is ground and the configuration $C' = (\{e\}, \langle\rangle)$ succeeds.

6. (Primitive Addition)

(a) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1, \cdots, g_{i-1}, g_{i+1}, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ i_1\ i_2\ i_3)$ and $i_1 + i_2 = i_3$

(b) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow}$ **failure**
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ i_1\ i_2\ i_3)$ and $i_1 + i_2 \neq i_3$

(c) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1\theta, \cdots, g_{i-1}\theta, g_{i+1}\theta, \cdots, g_a\theta\}, \langle \hat{e_1}\theta, \cdots, \hat{e_b}\theta \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ i_1\ i_2\ x)$ and $i_3 = i_1 + i_2$, where $\theta = \{x \mapsto i_3\}$

(d) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1\theta, \cdots, g_{i-1}\theta, g_{i+1}\theta, \cdots, g_a\theta\}, \langle \hat{e_1}\theta, \cdots, \hat{e_b}\theta \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ i_1\ x\ i_3)$ and $i_2 = i_3 \Leftrightarrow i_1$, where $\theta = \{x \mapsto i_2\}$

(e) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1\theta, \cdots, g_{i-1}\theta, g_{i+1}\theta, \cdots, g_a\theta\}, \langle \hat{e_1}\theta, \cdots, \hat{e_b}\theta \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ x\ i_2\ i_3)$ and $i_1 = i_3 \Leftrightarrow i_2$, where $\theta = \{x \mapsto i_1\}$

(f) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1\theta, \cdots, g_{i-1}\theta, g_{i+1}\theta, \cdots, g_a\theta\}, \langle \hat{e_1}\theta, \cdots, \hat{e_b}\theta \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ x_1\ 0\ x_3)$, where $\theta = \{x_1 \mapsto x_3\}$

(g) $(\{g_1, \cdots, g_i, \cdots, g_a\}, \langle \hat{e_1}, \cdots, \hat{e_b} \rangle) \overset{\text{res}}{\Leftrightarrow} (\{g_1\theta, \cdots, g_{i-1}\theta, g_{i+1}\theta, \cdots, g_a\theta\}, \langle \hat{e_1}\theta, \cdots, \hat{e_b}\theta \rangle)$
if $g_i \overset{\text{red}}{\Leftrightarrow} (\overline{plus}\ 0\ x_2\ x_3)$, where $\theta = \{x_2 \mapsto x_3\}$

7. Other arithmetic and comparison primitives are handled similarly.

Given a Cube program E, we can obtain its meaning by translating it into L_2, and determining $\text{Sol}(C_E)$. Each $(\{\}, \langle e_1, ..., e_n \rangle)$ in $\text{Sol}(C_E)$ represents one solution of the program, and can be visualized by transforming each e_i back into a term cube (as described on page 93), and filling these term cubes into the appropriate holder cubes in the original program.

A Cube computation may deadlock. This can happen for instance if we attempt to resolve an arithmetic predicate with an insufficient number of ground arguments. In the existing implementations, we indicate that a deadlock occurred, but we do not visualize *where* it occurred.

This could for instance be done by highlighting the predicate in the query in which the deadlock occurred.

Example 5.3.4 Consider the program shown in Figure 5.14. This program consists of the *List* type definition cube (also shown in Figure 5.6), a higher-order predicate *filter* (also shown in Figure 4.53), a predicate *even*, which succeeds whenever its only argument is even, and a goal *filter* applied to *even*, a list with elements 1 and 2, and an empty holder cube. Translating this program into L_0 yields:

$$
\begin{array}{l}
\Leftarrow \ \textbf{letrec} \\
\quad \textbf{type}\ \mathit{List}\{\mathsf{List}_1 = t_1\} = \mathit{nil} + \mathit{cons}\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : \mathit{List}\{\mathsf{List}_1 = t_1\}\}, \\
\quad \textbf{pred}\ \mathit{filter} = \lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}. \\
\qquad (\exists x_4, x_5 . x_4 = \mathit{nil} \wedge x_5 = \mathit{nil} \wedge x_4 = x_2 \wedge x_5 = x_3) \vee \\
\qquad (\exists x_6, \cdots, x_{14}. \\
\qquad\quad x_6 = \mathit{cons}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge \\
\qquad\quad x_9 = \mathit{cons}(\mathsf{cons}_1 = x_{10})(\mathsf{cons}_2 = x_{11}) \wedge x_1(\mathsf{in} = x_{12}) \wedge \\
\qquad\quad \mathit{filter}(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_{13})(\mathsf{filter}_3 = x_{14}) \wedge \\
\qquad\quad x_8 = x_{13} \wedge x_{11} = x_{14} \wedge x_2 = x_6 \wedge x_3 = x_9 \wedge x_7 = x_{12} \wedge x_7 = x_{10}) \vee \\
\qquad (\exists x_{15}, \cdots, x_{21}. \\
\qquad\quad x_{15} = \mathit{cons}(\mathsf{cons}_1 = x_{16})(\mathsf{cons}_2 = x_{17}) \wedge \\
\qquad\quad \mathit{filter}(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_{18})(\mathsf{filter}_3 = x_{19}) \wedge \\
\qquad\quad \mathit{not}(\mathsf{not}_1 = x_{21}) \wedge \\
\qquad\quad x_{15} = x_2 \wedge x_{19} = x_3 \wedge x_{17} = x_{18} \wedge x_{16} = x_{20} \wedge x_{21} = x_1(\mathsf{in} = x_{20})), \\
\quad \textbf{pred}\ \mathit{even} = \lambda\{\mathsf{even}_1 = x_{22}\}. \\
\qquad \exists x_{23}, x_{24}, x_{25}. \\
\qquad\quad \mathit{mod}(\mathsf{mod}_1 = x_{23})(\mathsf{mod}_2 = x_{24})(\mathsf{mod}_3 = x_{25}) \wedge x_{23} = x_{22} \wedge x_{24} = 2 \wedge x_{25} = 0 \\
\textbf{in}\ \exists x_{26}, \cdots, x_{34}. \\
\quad \mathit{filter}(\mathsf{filter}_1 = x_{26})(\mathsf{filter}_2 = x_{27})(\mathsf{filter}_3 = x_{28}) \wedge \\
\quad x_{26} = \mathit{even}\ (\mathsf{even}_1 \rightarrow \mathsf{in}) \wedge x_{27} = x_{29} \wedge x_{28} = x_{30} \wedge \\
\quad x_{29} = \mathit{cons}(\mathsf{cons}_1 = x_{31})(\mathsf{cons}_2 = x_{32}) \wedge x_{31} = 1 \wedge \\
\quad x_{32} = \mathit{cons}(\mathsf{cons}_1 = x_{33})(\mathsf{cons}_2 = x_{34}) \wedge x_{33} = 2 \wedge x_{34} = \mathit{nil}
\end{array}
$$

This program is quite inflated, as the translation algorithm presented in Section 5.1 is simple and straightforward, but produces lengthy expressions. Therefore, we compact the program — without changing its meaning — by resolving trivial unifications and eliminating unneeded variables:

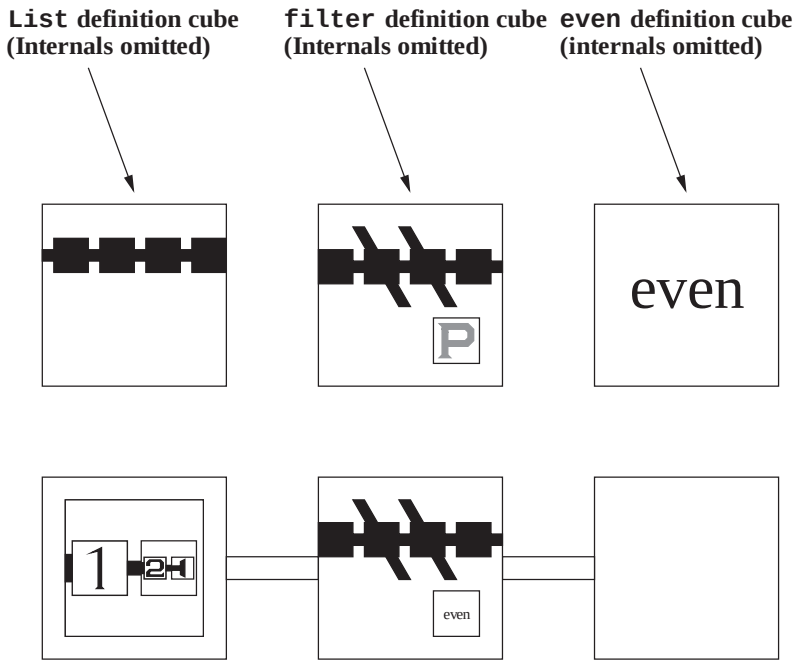


Figure 5.14: A Program Using "filter"

$\Leftarrow$ **letrec**

type $List\{\mathsf{List}_1 = t_1\} = nil + cons\{\mathsf{cons}_1 : t_1, \mathsf{cons}_2 : List\{\mathsf{List}_1 = t_1\}\}$,

pred $filter = \lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}.$

$(x_2 = nil \wedge x_3 = nil) \vee$

$(\exists x_4, x_5, x_6.$

$x_2 = cons(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge$

$x_3 = cons(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge x_1(\mathsf{in} = x_4) \wedge$

$filter(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee$

$(\exists x_7, x_8.$

$x_2 = cons(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$

$filter(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge$

$not(\mathsf{not}_1 = x_1(\mathsf{in} = x_7))),$

pred $even = \lambda\{\mathsf{even}_1 = x_9\}.mod(\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)$

in $\exists x_{10}.filter(\mathsf{filter}_1 = even\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 =$

$cons(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = cons(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = nil)))$

Translating this program into L_2 yields:

$$\Leftarrow \exists x_{10}.(\textbf{sel-1}\ \mathcal{F})(\mathsf{filter}_1 = (\textbf{sel-2}\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 = \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))$$

where $\mathcal{C}$ is $(\lambda\{\mathsf{cons}_1 = x_{12}, \mathsf{cons}_2 = x_{13}\}.\overline{cons}\ x_{12}\ x_{13})$ and $\mathcal{F}$ is

$$\begin{array}{l}
\textbf{fix}\, x_{11}\,.(2\textbf{-tuple} \\
\quad (\lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}. \\
\qquad (x_2 = \overline{nil} \wedge x_3 = \overline{nil}) \vee \\
\qquad (\exists x_4, x_5, x_6. \\
\qquad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge \\
\qquad\quad x_3 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge x_1(\mathsf{in} = x_4) \wedge \\
\qquad\quad (\textbf{sel-1}\ x_{11})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee \\
\qquad (\exists x_7, x_8. \\
\qquad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge \\
\qquad\quad (\textbf{sel-1}\ x_{11})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge \\
\qquad\quad (\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = x_1(\mathsf{in} = x_7)))) \\
\quad (\lambda\{\mathsf{even}_1 = x_9\}. \\
\qquad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
\qquad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)))
\end{array}$$

So, the initial configuration is $C_0 = (\{e_0\}, \langle x_{10}\rangle)$ with e_0 being

$$\begin{array}{l}
(\textbf{sel-1}\ \mathcal{F})(\mathsf{filter}_1 = (\textbf{sel-2}\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 = \\
\quad \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))
\end{array}$$

One possible sequence of resolution steps leading to a configuration in $\mathrm{Sol}(C)$ is the following:

- Select the goal e_0 from the configuration C_0.

$$
\begin{array}{ll}
e_0 = & (\textbf{sel-1}\ \textbf{fix}\, x_{11}\,.(2\textbf{-tuple} \\
& \quad (\lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}. \\
& \quad\quad (x_2 = \overline{nil} \wedge x_3 = \overline{nil}) \vee \\
& \quad\quad (\exists x_4, x_5, x_6. \\
& \quad\quad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge \\
& \quad\quad\quad x_3 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge x_1(\mathsf{in} = x_4) \wedge \\
& \quad\quad\quad (\textbf{sel-1}\ x_{11})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee \\
& \quad\quad (\exists x_7, x_8. \\
& \quad\quad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge \\
& \quad\quad\quad (\textbf{sel-1}\ x_{11})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge \\
& \quad\quad\quad (\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = x_1(\mathsf{in} = x_7)))) \\
& \quad (\lambda\{\mathsf{even}_1 = x_9\}. \\
& \quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
& \quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)))) \\
& (\mathsf{filter}_1 = (\textbf{sel-2}\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 = \\
& \quad \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil}))) \\
\overset{\text{red}}{\longmapsto}_{[FIX]} & (\textbf{sel-1}\ (2\textbf{-tuple} \\
& \quad (\lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}. \\
& \quad\quad (x_2 = \overline{nil} \wedge x_3 = \overline{nil}) \vee \\
& \quad\quad (\exists x_4, x_5, x_6. \\
& \quad\quad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge \\
& \quad\quad\quad x_3 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge x_1(\mathsf{in} = x_4) \wedge \\
& \quad\quad\quad (\textbf{sel-1}\ \mathcal{F})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee \\
& \quad\quad (\exists x_7, x_8. \\
& \quad\quad\quad x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge \\
& \quad\quad\quad (\textbf{sel-1}\ \mathcal{F})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge \\
& \quad\quad\quad (\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = x_1(\mathsf{in} = x_7)))) \\
& \quad (\lambda\{\mathsf{even}_1 = x_9\}. \\
& \quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
& \quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)))) \\
& (\mathsf{filter}_1 = (\textbf{sel-2}\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 = \\
& \quad \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))
\end{array}
$$

$$\overset{\text{red}}{\longmapsto}_{[SEL]} \quad (\lambda\{\mathsf{filter}_1 = x_1, \mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}.$$
$$(x_2 = \overline{nil} \wedge x_3 = \overline{nil}) \vee$$
$$(\exists x_4, x_5, x_6.$$
$$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge$$
$$x_3 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge x_1(\mathsf{in} = x_4) \wedge$$
$$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee$$
$$(\exists x_7, x_8.$$
$$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$$
$$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = x_1)(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge$$
$$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = x_1(\mathsf{in} = x_7))))$$
$$(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 =$$
$$\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))$$
$$\overset{\text{red}}{\longmapsto}_{[APP]} \quad (\lambda\{\mathsf{filter}_2 = x_2, \mathsf{filter}_3 = x_3\}.$$
$$(x_2 = \overline{nil} \wedge x_3 = \overline{nil}) \vee$$
$$(\exists x_4, x_5, x_6.$$
$$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge$$
$$x_3 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge$$
$$(\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_4) \wedge$$
$$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee$$
$$(\exists x_7, x_8.$$
$$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$$
$$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_3) \wedge$$
$$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_7))))$$
$$(\mathsf{filter}_3 = x_{10})(\mathsf{filter}_2 = \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))$$

$\overset{\text{red}}{\longmapsto}_{[APP]}$ $(\lambda\{\mathsf{filter}_2 = x_2\}.$

$(x_2 = \overline{nil} \wedge x_{10} = \overline{nil}) \vee$

$(\exists x_4, x_5, x_6.$

$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge$

$x_{10} = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge$

$(\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_4) \wedge$

$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee$

$(\exists x_7, x_8.$

$x_2 = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$

$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_{10}) \wedge$

$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_7))))$

$(\mathsf{filter}_2 = \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})))$

$\overset{\text{red}}{\longmapsto}_{[APP]}$ $(\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \overline{nil} \wedge x_{10} = \overline{nil}) \vee$

$(\exists x_4, x_5, x_6.$

$\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_5) \wedge$

$x_{10} = \mathcal{C}(\mathsf{cons}_1 = x_4)(\mathsf{cons}_2 = x_6) \wedge$

$(\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_4) \wedge$

$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_5)(\mathsf{filter}_3 = x_6)) \vee$

$(\exists x_7, x_8.$

$\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$

$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_{10}) \wedge$

$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_7)))$

We nondeterministically choose the third part of the overall disjunction, so $C_0 \overset{\text{res}}{\Leftrightarrow}_{[DISJ]} C_1 = (\{e_1\}, \langle x_{10} \rangle)$, where e_1 is

$\exists x_7, x_8.$

$\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \mathcal{C}(\mathsf{cons}_1 = x_7)(\mathsf{cons}_2 = x_8) \wedge$

$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_8)(\mathsf{filter}_3 = x_{10}) \wedge$

$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_7))$

- $C_1 \overset{\text{res}}{\Leftrightarrow}_{[EXIST]} C_2 = (\{e_2\}, \langle x_{10} \rangle)$, where e_2 is

$\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \mathcal{C}(\mathsf{cons}_1 = x_{18})(\mathsf{cons}_2 = x_{19}) \wedge$
$(\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_{19})(\mathsf{filter}_3 = x_{10}) \wedge$
$(\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_{18}))$

- $C_2 \overset{\text{res}}{\Leftrightarrow}_{[CONJ]} C_3 = (\{e_3, e_4, e_5\}, \langle x_{10} \rangle)$, where

  $e_3 = \mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) = \mathcal{C}(\mathsf{cons}_1 = x_{18})(\mathsf{cons}_2 = x_{19})$
  $e_4 = (\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = x_{19})(\mathsf{filter}_3 = x_{10})$
  $e_5 = (\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = x_{18}))$

- $\mathcal{C}(\mathsf{cons}_1 = 1)(\mathsf{cons}_2 = \mathcal{C}(\mathsf{cons}_1 = 2)(\mathsf{cons}_2 = \overline{nil})) \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (\overline{cons}\ 1\ (\overline{cons}\ 2\ \overline{nil}))$,
  $\mathcal{C}(\mathsf{cons}_1 = x_{18})(\mathsf{cons}_2 = x_{19}) \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (\overline{cons}\ x_{18}\ x_{19})$, and
  $(\overline{cons}\ 1\ (\overline{cons}\ 2\ \overline{nil}))$ and $(\overline{cons}\ x_{18}\ x_{19})$ unify with mgu $\theta = \{x_{18} \mapsto 1, x_{19} \mapsto (\overline{cons}\ 2\ \overline{nil})\}$.
  So $C_3 \overset{\text{res}}{\Leftrightarrow}_{[UNIF]} C_4 = (\{e'_4, e'_5\}, \langle x_{10} \rangle)$, where

  $e'_4 = (\mathbf{sel}\text{-}1\ \mathcal{F})(\mathsf{filter}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in}))(\mathsf{filter}_2 = (\overline{cons}\ 2\ \overline{nil}))(\mathsf{filter}_3 = x_{10})$
  $e'_5 = (\lambda\{\mathsf{not}_1 = x_{17}\}.\overline{not}\ x_{17})(\mathsf{not}_1 = (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1))$

- $e'_5 \overset{\text{red}}{\Leftrightarrow\!\!\!\!\rightarrow} (\overline{not}\ (\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1))$

  $(\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1)$ is ground. So, the negation resolution rule is applicable if the configuration $(\{(\mathbf{sel}\text{-}2\ \mathcal{F})\ (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1)\}, \langle\rangle)$ either succeeds or fails:

$$
\begin{array}{ll}
 & (\textbf{sel-2} \\
 & \quad (\textbf{fix}\, x_{11}\,. \\
 & \quad\quad (2\text{-}\textbf{tuple} \\
 & \quad\quad\quad (\cdots) \\
 & \quad\quad\quad (\lambda\{\mathsf{even}_1 = x_9\}. \\
 & \quad\quad\quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
 & \quad\quad\quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0))))) \\
 & (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1) \\
\overset{\text{red}}{\longmapsto}_{[FIX]} & (\textbf{sel-2} \\
 & \quad\quad (2\text{-}\textbf{tuple} \\
 & \quad\quad\quad (\cdots) \\
 & \quad\quad\quad (\lambda\{\mathsf{even}_1 = x_9\}. \\
 & \quad\quad\quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
 & \quad\quad\quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)))) \\
 & \quad\quad (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1) \\
\overset{\text{red}}{\longmapsto}_{[SEL]} & (\lambda\{\mathsf{even}_1 = x_9\}. \\
 & \quad\quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
 & \quad\quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)) \\
 & \quad\quad (\mathsf{even}_1 \rightarrow \mathsf{in})(\mathsf{in} = 1) \\
\overset{\text{red}}{\longmapsto}_{[CAST]} & (\lambda\{\mathsf{in} = x_9\}. \\
 & \quad\quad\quad (\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
 & \quad\quad\quad\quad (\mathsf{mod}_1 = x_9)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)) \\
 & \quad\quad (\mathsf{in} = 1) \\
\overset{\text{red}}{\longmapsto}_{[APP]} & ((\lambda\{\mathsf{mod}_1 = x_{14}, \mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ x_{14}\ x_{15}\ x_{16}) \\
 & \quad\quad (\mathsf{mod}_1 = 1)(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)) \\
\overset{\text{red}}{\longmapsto}_{[APP]} & ((\lambda\{\mathsf{mod}_2 = x_{15}, \mathsf{mod}_3 = x_{16}\}.\overline{mod}\ 1\ x_{15}\ x_{16})(\mathsf{mod}_2 = 2)(\mathsf{mod}_3 = 0)) \\
\overset{\text{red}}{\longmapsto}_{[APP]} & ((\lambda\{\mathsf{mod}_3 = x_{16}\}.\overline{mod}\ 1\ 2\ x_{16})(\mathsf{mod}_3 = 0)) \\
\overset{\text{red}}{\longmapsto}_{[APP]} & (\overline{mod}\ 1\ 2\ 0)
\end{array}
$$

$(\{(\overline{mod}\ 1\ 2\ 0)\}, \langle\rangle) \overset{\text{res}}{\Leftrightarrow}_{[PRIM]}$ **failure**. So $C_4 \overset{\text{res}}{\Leftrightarrow}_{[NOT]} C_5 = (\{e'_4\}, \langle x_{10}\rangle)$.

At this point, each reduction and each resolution rule was used at least once. Therefore, we do not show the rest of the resolution process in detail. The only resolution sequence from C_0 to

a success-configuration is

$$C_0 \overset{\text{red}}{\Leftrightarrow} \cdots \overset{\text{red}}{\Leftrightarrow} C_5 \overset{\text{red}}{\Leftrightarrow} \cdots \overset{\text{red}}{\Leftrightarrow} C_{FIN} = (\{\}, \langle(\overline{cons}\ 2\ \overline{nil})\rangle)$$

All other sequences either lead to **failure**, or do not terminate. So $\mathrm{Sol}(C_0) = \{C_{FIN}\}$.

C_{FIN} is visualized by filling the holder cube associated with x_{10} with the term cube representing $(\overline{cons}\ 2\ \overline{nil})$.

Chapter 6

Implementation

Currently, there are two implementations of a Cube environment. The first implementation, called CUBE-I, served as a feasibility study and as a test-bed for trying out various language design choices, concerning both the visual appearance of constructs and their semantic behavior. In order to be able to explore out new ideas quickly, we chose to implement the bulk of the first system in Lazy ML [2], which greatly sped up the development.

The two main limitations of the first system were low speed and the almost complete absence of an interactive user interface. The second implementation of Cube, CUBE-II, improves on both of these shortcomings.

The next two subsections contain more details about the capabilities and the general architecture of these two implementations.

6.1 The First Implementation

The CUBE-I system is able to read in a prefabricated program, infer its type and the types of its subexpressions, evaluate it, and render the program, the inferred types, and the computed values onto an X window. It allows the user to navigate through the program; however, it does not support any editing actions. It also lacks interactive features such as an option to load other programs, to change the colors of objects, etc.

The system consists of two programs: The Front-End, a C program responsible for rendering and for mouse interaction, and the Back-End, a Lazy ML program responsible for everything

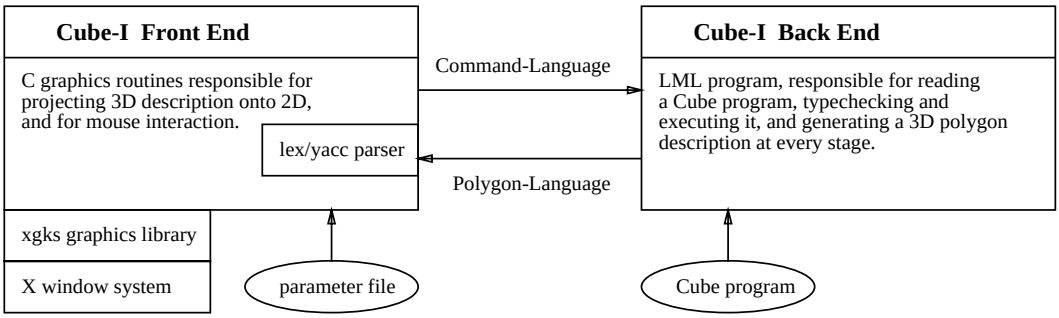


Figure 6.1: Block Diagram of the Prototype System

else. The two programs run concurrently, and communicate over Unix streams. Figure 6.1 shows a block diagram of the different components of the system.

We implemented the system in such a way in an attempt to achieve fast rendering on one hand, and easy and rapid system development on the other. The rendering step is the performance bottleneck, so it was mandatory to implement it in a fast, low-level language, such as C. On the other hand, the rendering routines comprise less than a quarter of the code, the less time-critical parts could still be implemented in a high-level language such as Lazy ML, whose advanced features greatly sped up development time.

The Front-End displays a Cube program either as a wireframe rendering (see Figure 6.2), or it uses a more complex technique, which not only performs hidden-surface removal, but is also able to deal with transparent surfaces (see Figure 6.3). The user can toggle between the two rendering qualities with the button labeled "FLIP".

The high-quality rendering technique works as follows: The Front-End receives a set of colored, and possibly transparent polygons in 3-space, which may contain holes. It performs the appropriate translation, rotation, and scaling operations, and then computes which pixels each polygon covers, using a scan conversion algorithm (see for instance [28]). For each covered pixel, it records the color (denoted by a triple (r, g, b) representing the spectral components), transparency (denoted by a coefficient α), and z-coordinate. As there may be several transparent polygons covering the same pixel, the renderer needs to retain a list of (r, g, b, z, α) values for each pixel.

After the polygons have been rasterized, the list of (r, g, b, z, α) values of each pixel is first sorted by z value and then blended together from back (high z) to front (low z). This results

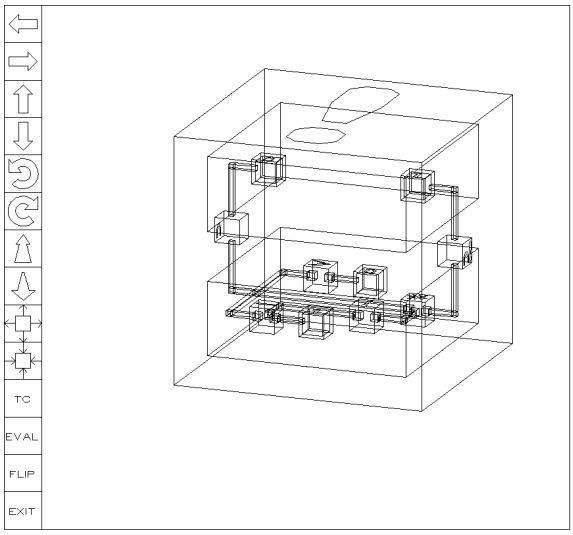


Figure 6.2: CUBE-I wireframe rendering of the factorial predicate

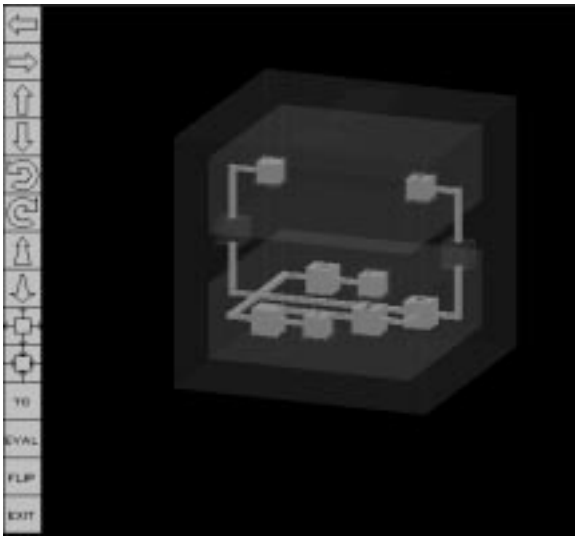


Figure 6.3: CUBE-I high-quality rendering of the factorial predicate

in a single (r, g, b) triple for each pixel. These (r, g, b) values are then drawn onto the rendering area of the window.

The first implementation uses XGKS [70], the X windows implementation of the Graphics Kernel System [33], to perform the actual drawing. Due to inefficiencies in the implementation of XGKS, establishing a high-resolution picture covering 477×477 pixels takes about 30 seconds.

The Front-End can send three types of messages to the Back-End: "loadFile *foo*", "typeCheck", and "execute". The Back-End, in turn, replies to each request of the Front-End by sending it a list of colored polygons in 3-space, describing the new scene, and then waits for the next request. The system is thus driven by the Front-End.

Upon startup of the system, the Front-End parses the command-line arguments, expecting the name of a file describing a Cube program, and then asks the Back-End to load this program. Upon such a "loadFile *foo*" request, the Back-End reads in a structured picture description (dubbed HLPD, or "High-Level Picture Description") from the file *foo*, converts it into an unstructured set of polygons, and hands those to the Front-End.

The Front-End also controls the "buttons" on the left of the drawing area. The top eight buttons (rotate left/right/up/down/counterclockwise/clockwise, move forward/backward, zoom in/out) control the position of the "camera"; these requests are handled by the Front-End di-

rectly. The "FLIP" button toggles between wireframe and high-quality rendering, this request can also be handled directly by the Front-End. The buttons labeled "TC" and "EVAL" trigger type inference and evaluation, the Front-End sends the appropriate message to the Back-End.

Upon a "typeCheck" request, the Back-End converts the structured picture description into a more textual form (very similar to L_0), performs type inference on this textual form, visualizes the inferred types, translates the structured picture description and the visualized types into a set of polygons, and transmits those back to the Front-End.

Upon an "execute" request, it converts the structured picture description into a textual form similar to L_2, and then evaluates this textual program. Evaluation is conceptually performed in parallel; we realized this by maintaining queues of "processes" and "threads", performing one resolution step on a thread, and then selecting a new thread in a new process. The result of the computation — provided it terminates — is then mapped back into a visual form, translated together with the initial program into a set of polygons, and transmitted to the Front-End.

The "QUIT" button terminates the program; the Front-End closes the stream connecting it to the Back-End, and then terminates. The Back-End, noticing that its incoming communication channel has been closed, then terminates as well.

6.2 The Second Implementation

The second implementation of the Cube system consists of a single program, written entirely in Modula-3 [60]. We chose this language, as it is almost as efficient as C, but at the same time offers a rich set of features that make development much easier. Modula-3 is an offspring of Modula-2. It offers modules, object-oriented features, generics (known as "templates" in C++), exceptions, preemptive multitasking, and garbage collection[1]. It also comes with an extensive library of existing modules and classes. In particular, it supplies a multi-threaded, object-oriented windowing system, Trestle [48], which is implemented on top of X Windows. Trestle is complemented by a hierarchy of window abstractions and a rich widget set containing buttons, scrollers, filebrowsers, etc. [4, 5]. CUBE-II's entire user interface is based on this widget set.

[1] Modula-3 could be described as "C++ plus modules plus threads plus garbage collection minus multiple inheritance"

Figure 6.4: CUBE-II Evaluation Control Panel

Reimplementing the first Cube system in Modula-3 greatly increased the performance of the system, in particular of the components which were located in the former Back-End. Abandoning the use of XGKS and instead using the X Window System [73] directly brought along another dramatic increase in speed. Rendering a picture containing a few hundred polygons onto an area of 640×512 pixels takes now around 10 seconds.

In the first system, the user had to press a button to switch from wireframe to high-quality rendering, and was then unable to interact with the system for some 30 seconds, until the rendering was complete. In the new system, a change in the scene or in the camera-position causes the rendering area to be immediately updated by a wireframe representation of the new scene. At the same time, a dedicated rendering thread starts to compute a high-quality picture in the background. If the scene changes before the rendering thread has completed its task, it is alerted by the thread which caused the change, and restarts its computation. Otherwise, upon successful completion, it replaces the wireframe rendering by a high-quality picture.

The result of this approach is that the user never has to wait for a high-quality rendering to complete, but can constantly interact with the system. If he remains idle long enough, the high-quality rendering appears automatically.

Multi-threading is also used by the Cube interpreter to deal with infinite computations. When the user presses the "EVAL" button, a separate thread is created to perform the evaluation. At the same time, a control panel (see Figure 6.4) is popped up, which informs the user how many solutions have been found so far, and allows him to interrupt the evaluation. In addition, this control panel allows him to browse through the various solutions. Figure 6.5 shows a program which will generate all the natural numbers. Figure 6.6 shows the user inspecting the

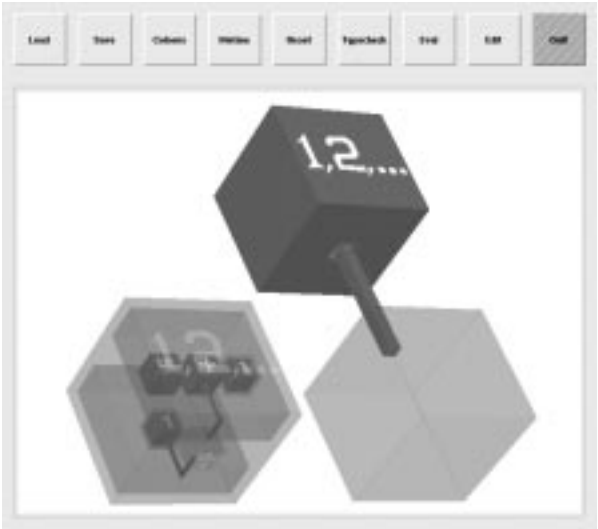


Figure 6.5: A Program for Computing All the Natural Numbers

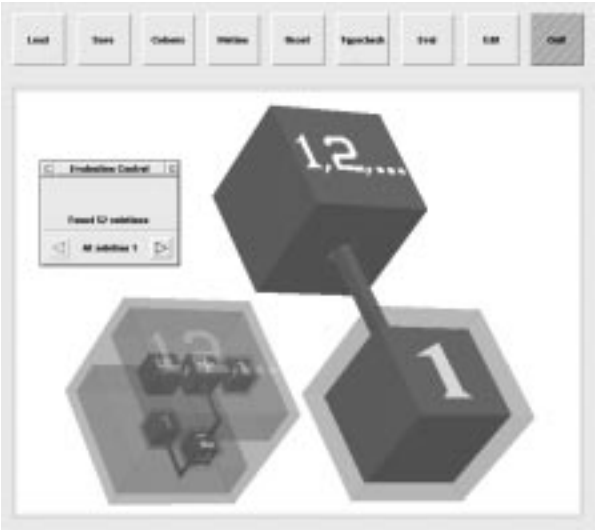


Figure 6.6: Program From Figure 6.5 Displaying the First Solution

Figure 6.7: Program From Figure 6.5 Displaying the Second Solution

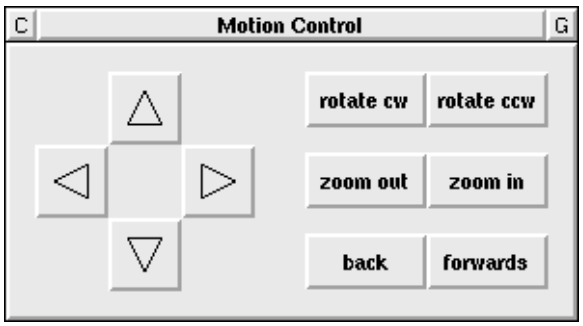


Figure 6.8: CUBE-II Motion Control Panel

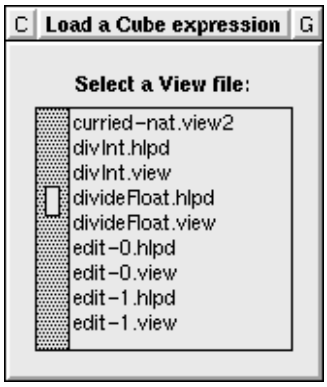


Figure 6.9: CUBE-II Load File Menu

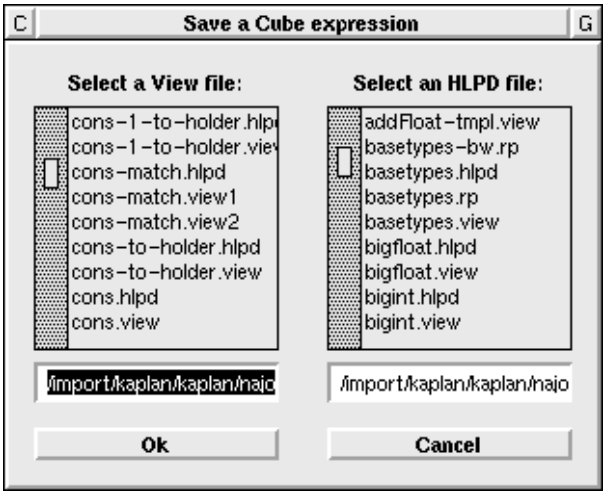


Figure 6.10: CUBE-II Save File Menu

first solution, Figure 6.7 shows him inspecting the second one, having aborted the computation after 52 solutions have been found.

6.2.1 The User Interface

The main window of the new system consists of a menu bar at the top and the rendering area below. The menu bar offers nine different options: "LOAD", "SAVE", "COLOURS", "MOTION", "RESET", "TYPECHECK", "EVAL", "EDIT", and "QUIT".

"TYPECHECK", "EVAL", and "QUIT" do the obvious things. The "RESET" option removes inferred types and computed values from the visualized program. The "MOTION" button pops up a motion control panel, which provides controls for moving the "camera" through the scene shown in the rendering area. Figure 6.8 shows the motion control panel.

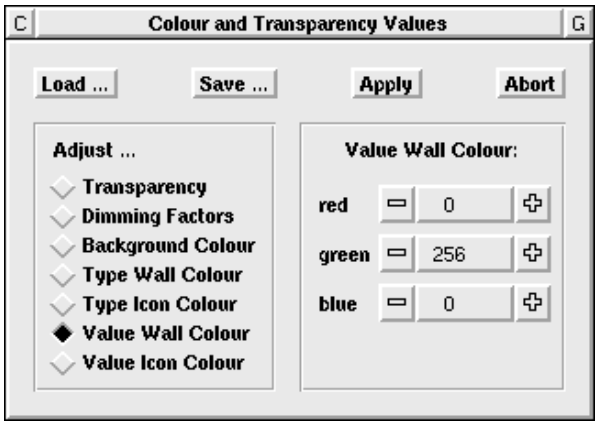


Figure 6.11: CUBE-II Rendering Control Panel

"LOAD" and "SAVE" provide options to load existing programs, and to save new or modified ones. A program description consists of two parts: The program itself (usually stored in a file with the suffix `.hlpd`) and a description of the camera's relevant parameters (location, orientation, zoom factor, etc.), which is usually stored in a file with the suffix `.view`. The view file contains a reference to the HLPD file, so in order to load a program, the user loads a view file, which will then automatically load the corresponding HLPD file. Figure 6.9 shows the menu used to select a view file. In order to save a program description, the user has to specify both view file and HLPD file (see Figure 6.10).

The "COLOURS" button pops up a menu (see Figure 6.11) which allows the user to manipulate the important parameters used by the high-quality renderer: transparency coefficient, direction and intensity of light sources, and color values of the different objects, such as types, values, pipes, icons, etc. It also allows him to save his customizations of these rendering parameters to a file, and to load them back again. When the Cube system is started, it will look for a file `.cube-renderparams` in the current working directory, and if it exists, interpret it as a rendering parameter file. This mechanism is intended to allow users to customize some aspects of Cube's visual representation to their liking.

The "EDIT" option, finally, allows the user to edit existing programs, or to create new ones. Due to time limitations, we were unable to build a full-fledged editor that supports every syntactic construct; however, the crucial operations are supported.

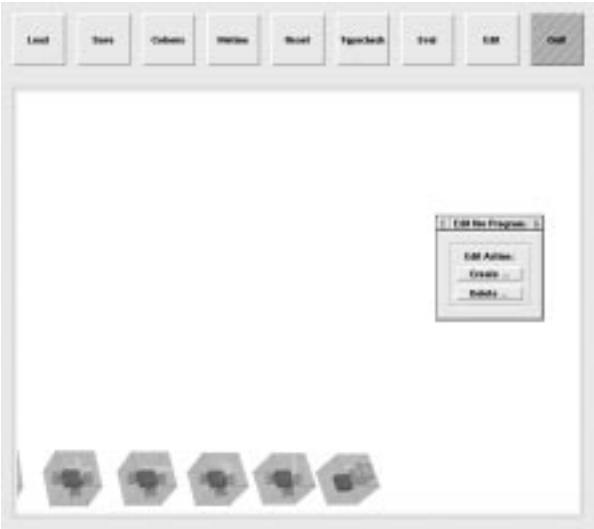


Figure 6.12: Selecting the Create Option

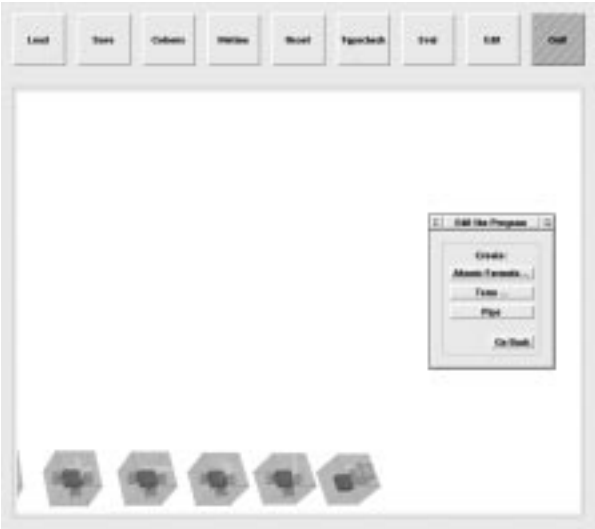


Figure 6.13: Selecting the Atomic Formula Option

6.2.2 The Editor

Editing a three-dimensional program on a two-dimensional screen is an interesting and difficult problem. A two-dimensional pointing device, such as a mouse, can only be used to specify a point in 2-space, which translates to a line rather than a point in 3-space. This means that either the user must perform two pointing operations to completely specify a point in 3-space, or the system has to use information obtained from the user's intent and the structure of the existing picture to select one particular point on the line.

The following example shall illustrate how editing works in the new system. Assume that we want to construct an expression that performs temperature conversions between Celsius and Fahrenheit (see page 41).

First, we want to create a holder cube that shall contain the temperature value in Celsius. A holder cube could be located anywhere; specifying a line as opposed to a point in 3-space is not sufficient, the system would be unable to use any information based on the structure of the existing picture to determine the right point on the line. Hence we have to supply more location information than just a simple mouse-click. We press the "EDIT" button, select the "CREATE" option from the *Edit* menu (Figure 6.12), the "ATOMIC FORMULA" option from the *Create* submenu (Figure 6.13), and the "VALUE HOLDER CUBE" option from the *Create Atomic Formula* submenu (Figure 6.14). Now we locate a point in the rendering area and press

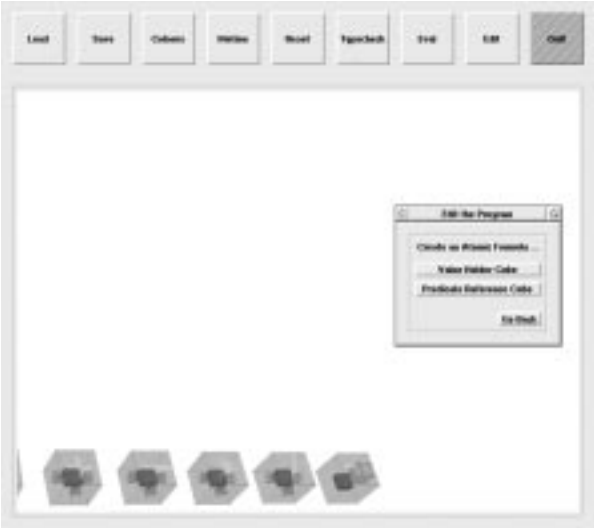


Figure 6.14: Selecting the Value Holder Cube Option

Figure 6.15: Selecting a Point on the Screen

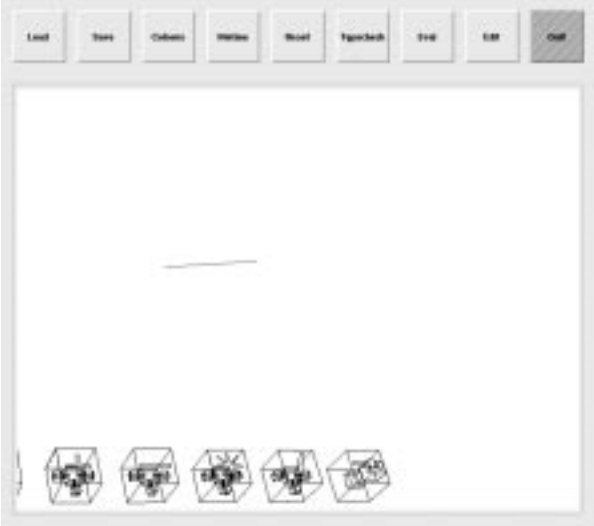


Figure 6.16: Rotating the Scene

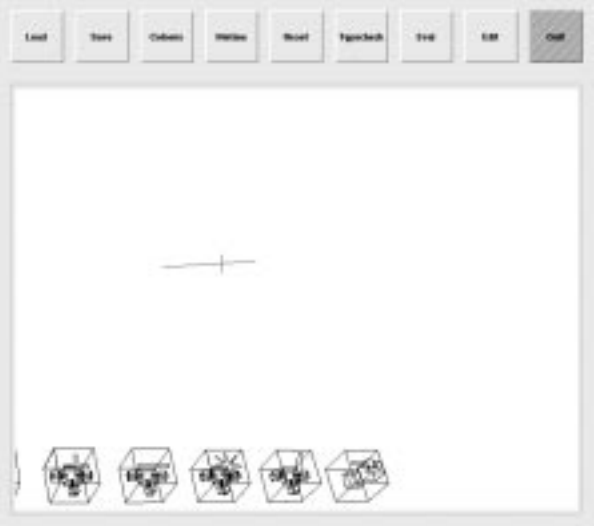


Figure 6.17: Selecting a Point on the Ray

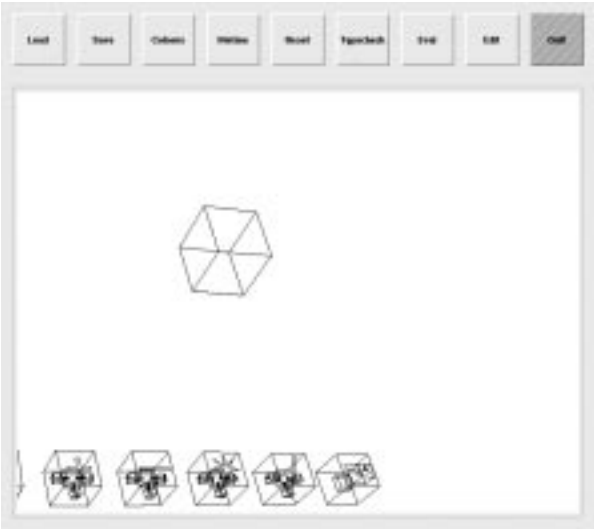


Figure 6.18: Fixing the Size of the Cube

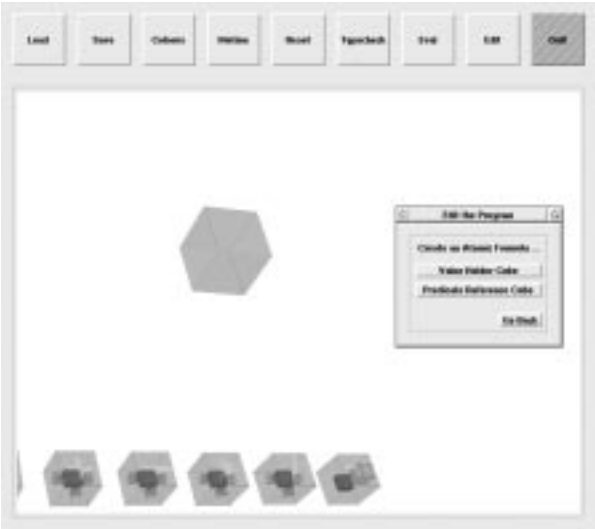


Figure 6.19: Selecting the Create Predicate Reference Cube Option

down the left mouse button (Figure 6.15), thereby specifying a line in 3-space. We drag the mouse to the left or to the right; the scene will rotate around a point on the line we just selected, and the line — highlighted in red — will thus become visible (Figure 6.16). Once we release the left mouse button, the rotation stops, and a vertical bar (highlighted in red) appears, which slides along the line we just specified and tracks the x position of the mouse (Figure 6.17). The point where bar and line cross describes a unique 3D location. We move the bar to the point on the line that we want, and finish the selection process by clicking the left mouse button.

The selected point is taken to be the center of the holder cube that we want to construct. Line and bar disappear, and get replaced by the holder cube. We move the mouse away from the cube's center to increase its size (Figure 6.18). Clicking the left mouse button fixes the size and finishes the holder cube construction process.

Next we would like to create a reference cube referring to the floating-point multiplication predicate. We work our way through various submenus to the "Create Predicate Reference Cube" option and select it (Figure 6.19). Now we need to specify which predicate we would like to refer to. This can be done by simply locating the predicate definition cube defining the primitive floating-point multiplication predicate in the rendering area and clicking on it. The 2D point translates into a 3D line, but the system can use structural information

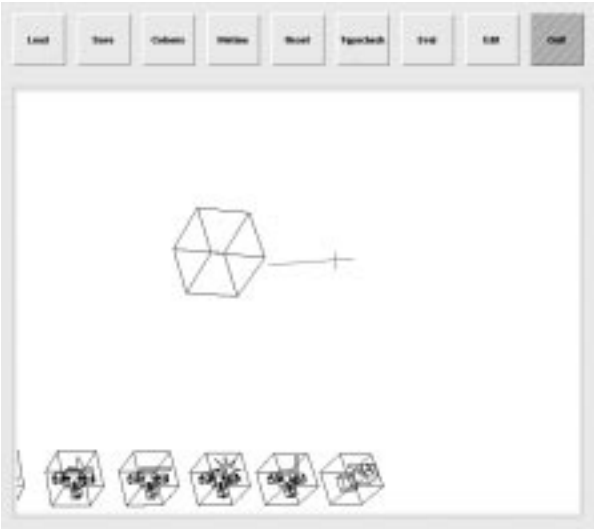


Figure 6.20: Specifying a 3D Point

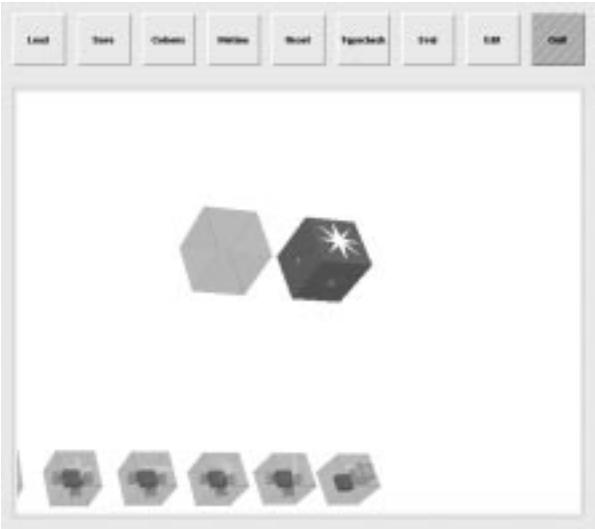


Figure 6.21: The Predicate Cube Appears

— the existing predicate definition cubes — and the user's intent — selecting such a cube — to determine which object on the line to select.

Having selected the predicate we would like to refer to, we need to position the reference cube in 3-space. This operation is similar to positioning a holder cube; the reference cube could be positioned anywhere. We have to go through the same motions as before: locate a point in the rendering area, press down the left mouse button to specify a line going through this point, move the mouse to rotate the scene, release the button to terminate rotation and make the intersecting bar appear (Figure 6.20), position the bar on the desired point on the line, and click the left mouse button to terminate the 3D point selection process. The specified point is taken to be the center of the new reference cube, the cube appears around it (Figure 6.21), its size is the same as that of the definition cube it refers to.

We perform similar actions to create a reference cube referring to the floating-point addition predicate and three more holder cubes (Figure 6.22).

Next we would like to connect the various holder cubes and ports through pipes. There are many techniques imaginable that Cube could use for pipe construction, from asking the user to specify the end points and every single joint of a pipe through 3-space point selection (this would be easy to implement, but would place a heavy burden on the user) to just asking the user to specify the end-points (which can be done with a single mouse-click per end-point, as

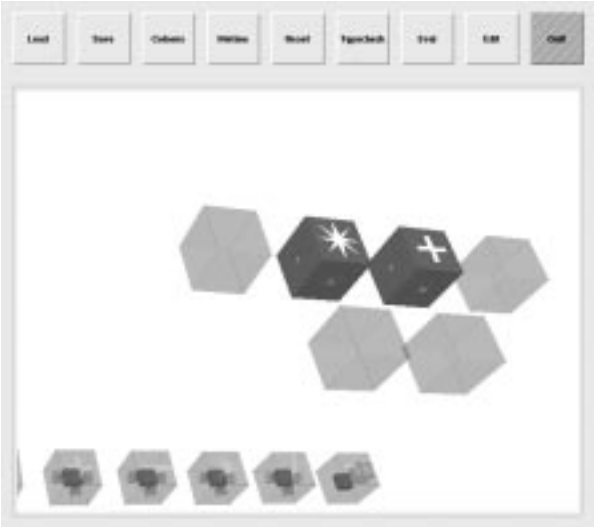


Figure 6.22: All the Holder and Predicate Cubes Have Been Created

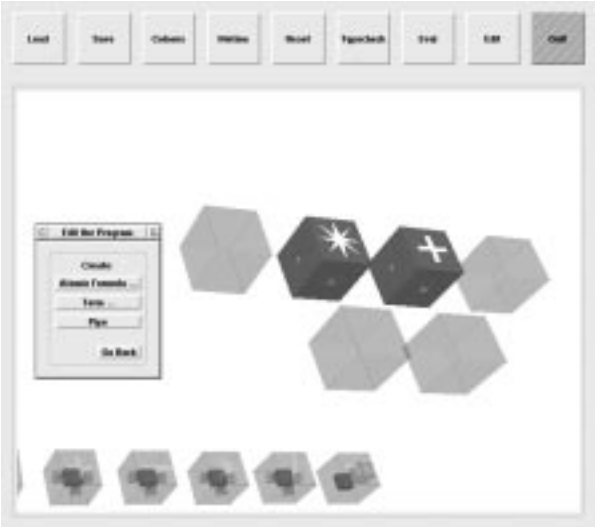


Figure 6.23: Selecting the Create Pipe Option

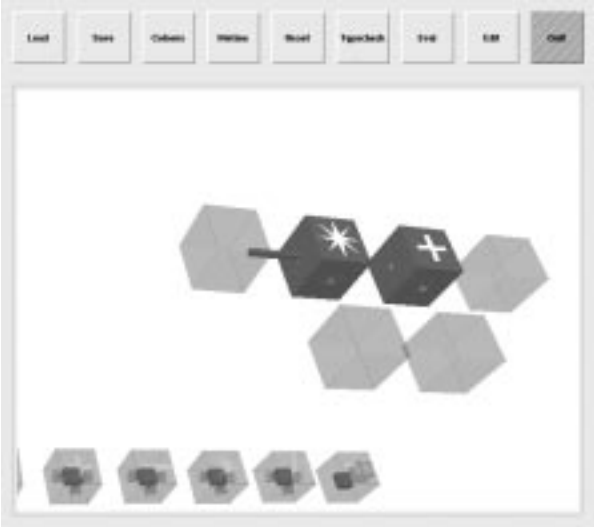


Figure 6.24: The Pipe Has Appeared

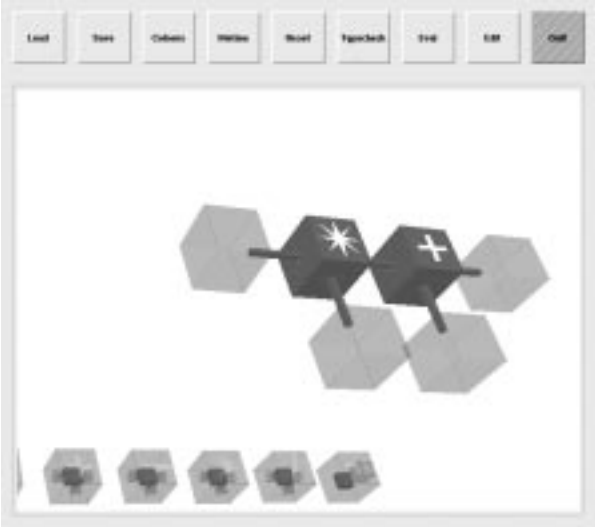


Figure 6.25: All the Pipes Have Been Created

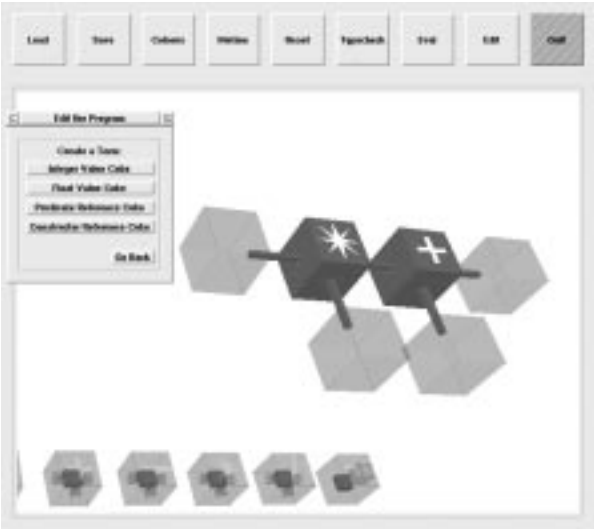


Figure 6.26: Selecting the Create Floating-Point Term Cube Option

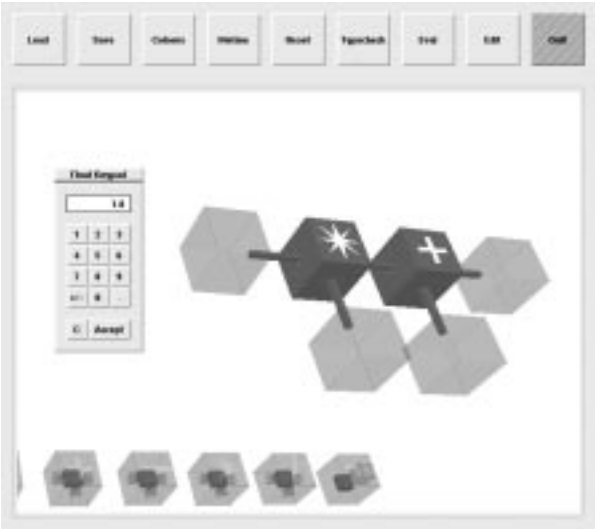


Figure 6.27: Typing in the Value 1.8

the system can use structural information) and then routing the pipe automatically, avoiding obstacles while minimizing its length and the number of joints.

The current implementation represents a compromise: The user has to specify the end-points of a pipe, simply by clicking on them; the system can use structural information to determine which objects that could serve as pipe termini are located on the 3D line corresponding to the 2D point supplied by the user. The system then attempts to route a pipe between those two termini. It will avoid routing the pipe through another "solid" object, but it will not try to achieve a visually pleasing routing. Of course, in many of the simpler cases this is entirely sufficient.

In our example, we select the "CREATE A PIPE" option from the appropriate submenu (Figure 6.23), then click on the leftmost holder cube, thereby specifying one terminus of the pipe, and on the port representing the first argument to the addition cube, thereby specifying the second terminus. The system now connects those two termini through a pipe (Figure 6.24). We repeat the process, until all the ports are connected either to another port or to a holder cube (Figure 6.25).

Finally, we want to place a floating-point constant (namely 1.8) into the holder cube connected to the second argument of the multiplication predicate. We select the "CREATE FLOATING-POINT TERM CUBE" option (Figure 6.26), use the mouse to locate the holder cube

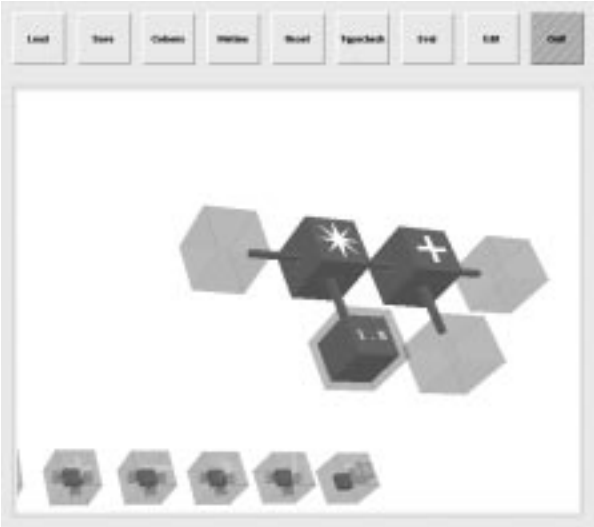


Figure 6.28: The Floating-Point Term Cube Appeared

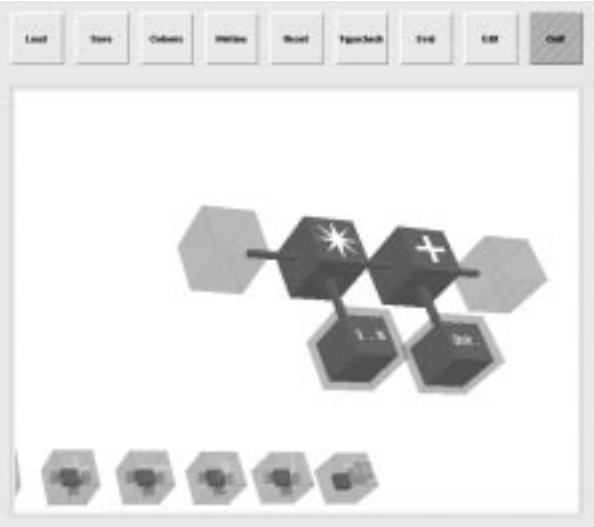


Figure 6.29: Creating the Term Cube Representing 32.0

Figure 6.30: Creating the Term Cube Representing 20.0

Figure 6.31: After Type Inference

Figure 6.32: After Evaluation

Figure 6.33: Selecting the Delete Term Cube Option

in the rendering area, and click on it. The system knows that we are trying to select an empty holder cube, and it determines which such cube lies on the 3D line corresponding to the 2D point we clicked on. A floating-point keyboard pops up, and we type in the number 1.8 (Figure 6.27). Now a floating-point term cube referring to the constant 1.8 appears inside the holder cube (Figure 6.28).

We use the same approach to place the constant 32.0 into the holder cube connected to the second argument of the addition predicate (Figure 6.29). This leaves us with a complete temperature conversion program.

To convert from Celsius to Fahrenheit, we place a floating-point value — say 20.0 — into the left empty holder cube (Figure 6.30). Type inference tells us that the right holder cube will receive a floating-point value (Figure 6.31), and pressing the "EVAL" button fills it with the right value, namely 68.0 (Figure 6.32).

In order to perform another computation, we have to "reset" the program, and delete the value 20.0 from the leftmost holder cube. We do this by selecting the "DELETE TERM CUBE" option (Figure 6.33), locating the floating-point term cube in the rendering area, and clicking on it. The system knows that we want to delete an existing term; it determines which term lies on the line we just specified, and removes it from the program (Figure 6.34).

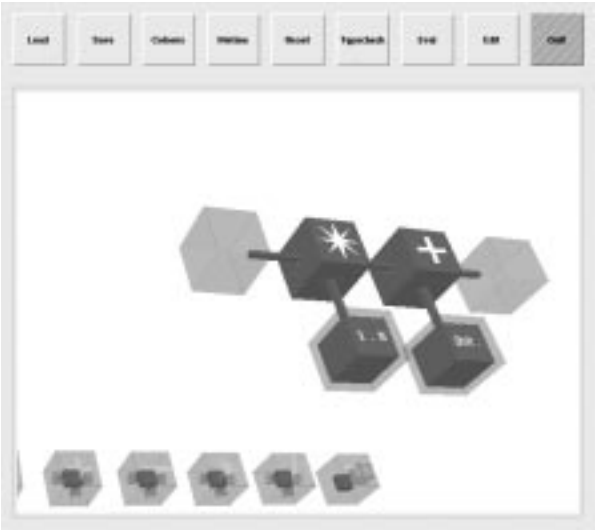


Figure 6.34: Having Deleted the Term Cube 20.0

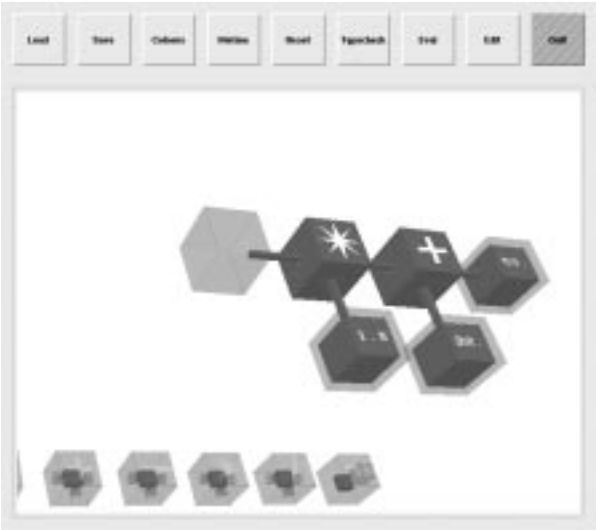


Figure 6.35: Creating the Term Cube 50.0

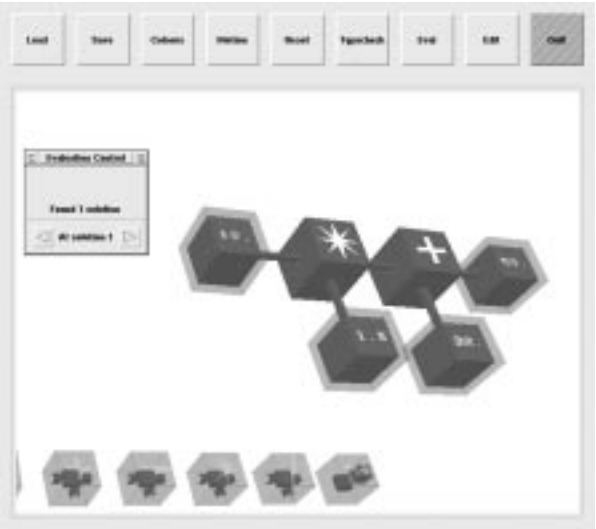


Figure 6.36: After Evaluation

In order to convert from Fahrenheit to Celsius, we place a floating-point value — say 50.0 — into the right empty holder cube (Figure 6.35) and press "EVAL". This time, the leftmost holder cube receives a result, namely 10.0 (Figure 6.36).

Creating this temperature conversion program takes a skilled user a few minutes — for sure longer than it would take someone to specify the same program in a high-level textual language. However, much of the tediousness of program construction can be attributed to the fact that Cube currently uses a 2D window system and a 2D pointing device. A virtual-reality-based programming environment will certainly speed up the editing process considerably. The remaining editing overhead to textual languages will be made up (or so we hope) by the greater intuitiveness and better debugging features of those future 3D visual languages.

Chapter 7

Conclusion

Our work has been an exploratory foray into the use of three-dimensional graphics for visual programming. We have developed Cube, a 3D visual language, and we have demonstrated the feasibility of the underlying ideas by building two prototype implementations of the language. Our contributions to the field of visual programming can be summarized as follows:

- Cube is the *first full-fledged three-dimensional visual language*. For the last 6 years, there has been interest in the visual language community regarding 3D languages [24], and speculations about their potential. But no such languages have been developed prior to Cube. However, there have been subsequent designs of 3D visual languages, e.g. Lingua Graphica [81] and CAEL-3D [68].

- The third dimension can be used to provide a *syntactically richer language*, much like color, texture, distance, directions, and spatial enclosure have been used before. We use 3D to encode *semantic information*: We represent terms, atomic logic formulas, and types by cubes; we express conjunction of atomic formulas and product of types by arranging the corresponding cubes horizontally to each other (i.e. in the xy plane); and we express disjunction of conjoint formulas and sum of tagged product types by arranging the corresponding horizontal planes vertically, in the z dimension (i.e. by stacking them on top of each other). The fact that we have two dimensions — x and y — to denote conjunctions allows us to connect the various cubes which form the conjunction through pipes, i.e. a data flow diagram. Data flow diagrams are inherently an (at least) two-dimensional formalism, the fact that we have a third dimension at our disposal helps us in avoiding

the problem of crossing lines — we can always route a pipe in 3-space so that it does not collide with any other pipe.

- A three-dimensional syntax opens up the possibility of utilizing a virtual-reality-based programming environment. Programming in Virtual Reality is interesting in its own right, due to the immersive and reactive nature of virtual realities. VR environments improve the input component of graphical user interfaces by allowing for direct manipulation: instead of using a mouse to interact with an object on the screen, the user can handle a virtual object directly, by mediation of a data glove. Moreover, selecting the focus of attention is simplified: in a 2D window system, the user is confronted with many windows; he has to use the mouse to select a window. In a virtual reality environment, on the other hand, all objects occupy the same virtual world; the user focuses on a particular object simply by looking at it and manipulating it directly. Finally, in the 2D setting, if the virtual canvas of a window is larger than the physical window, the user has to perform panning operations to access hidden parts of the window. This is usually done either by using scrollbars, or by "dragging" the canvas with the mouse. In a Virtual Reality setting, however, panning is replaced by simple head movement.

  Another situation in which a 3D language is useful is when the application domain of the language deals with three-dimensional representations, such as 3D animations or virtual reality applications. CAEL-3D [68] was developed to support the development of 3D animations, and Lingua Graphica [81] is aimed at supporting the development of virtual-reality applications.

- Cube shows how to incorporate a static polymorphic type system into a visual language. The benefits of static type systems are widely recognized: they help in detecting programming errors statically, without the need to perform exhaustive run-time testing. We use the Hindley-Milner algorithm to ensure the well-typedness of programs. The Hindley-Milner algorithm does not rely on user-supplied type declarations, instead, it infers (or reconstructs) the types of the expressions of a program. We provide additional feedback to the user by visualizing the inferred types of variables. In this respect, Cube is superior to most textual languages that use Hindley-Milner; these languages typically just indicate whether or not the program is well-typed.

Cube guarantees that a well-typed program is type-safe, i.e. that it will not fail at run-time due to a type error. We were the first to propose the use of Hindley-Milner type inference for visual languages and to make strong guarantees about type-safety [56]. Our work has influenced several other visual languages [8, 65, 66].

- Cube is based on Horn logic, a powerful, declarative formalism. Horn logic was first proposed as a programming language by Robert Kowalski in the 1970's [40]. Prolog [13], jointly developed by him and Alain Colmerauer, is the most widespread logic programming language to date. Logic programming is an attractive paradigm due to its declarative nature, its inherent parallelism (in form of AND and OR nondeterminism), and its multi-directional nature (predicates have no input/output patterns, and logic variables can be viewed as multidirectional communication channels).

  One problematic feature of logic languages such as Prolog is that programs are represented by a flat set of clauses; there are no mechanisms to localize definitions. Cube solves this problem by allowing for nested predicate definitions. It also eliminates Prolog's unclean features, such as the "cut" predicate (a mechanism to prune the search space, which sacrifices completeness), its dependency on clause and subgoal orderings, or its depth-first-search resolution strategy. Cube exploits the implicit parallelism of logic programs by using a pseudo-concurrent interpreter; concurrency is simulated via time-slicing.

- Cube is a higher-order programming language. It treats predicates (the "agents of computation") as first-class objects, and allows them to be passed as arguments to other predicates. A predicate which takes another predicate as an argument is called higher-order. In the functional programming community, higher-orderness has been identified as one of the most powerful abstraction mechanisms, which can leads to a very high degree of code reuse.

  Cube's notion of higher-orderness is simplistic (we use an intensional notion of equality for predicates), but it is powerful enough to allow us to build the higher-order abstractions common in functional programming; moreover, it is implemented very efficiently.

- Cube applies the visual data flow metaphor to logic programming. Cube uses data flow to denote "shared variables". This idea would appear to be obvious; in fact, in the area

of concurrent logic programming, logic variables are often referred to as "communication channels", a term which even more so evokes the mental image of a link connecting two parties. However, we are not aware of any other visual programming language which is based on logic and uses the data flow metaphor. Consequently, these other languages have been rather weak in describing variable sharing: Some of them (like Senay and Lazzeri's system [74]) use textual symbols to identify variables, others (like VLP [41]) use "shared patterns", i.e. an iconic approach. Either way, no visual aid is provided to the user in noticing interconnections.

- Finally, Cube demonstrates that there is no inherent performance penalty to visual languages. We show this by defining an isomorphic textual language, translating visual programs into their textual counterparts, and performing all computations on the textual structure. The only price we pay for using a visual notation is the overhead of mapping visual programs into a textual form, and mapping the results back into pictures. This overhead is constant, that is, it does not depend on the length of the actual computation.

We see several possible directions for future work. The most obvious omission in this thesis is a Cube environment that is virtual-reality-based. Such an environment would alleviate much of the tediousness which is currently associated with the construction of Cube programs. It is a definite requirement for constructing programs that are larger than the toy programs we have shown here.

The second major omission of this thesis is an evaluation of the *usefulness* of Cube. Ultimately, the usefulness of a programming language can be determined only through empirical studies, either by conducting an experiment that compares how well a number of test subjects can solve a problem or a set of problems in different languages, or by using the language to complete a significant programming project, and then analyzing how fast the task could be completed, and how correct, how maintainable, and possibly how reusable the resulting program is. Both approaches would have required a more comfortable programming environment than the one we were able to build with the technology available to us.

A second promising direction for future research lies in *providing customizable visual representations of values*. For example, a two- or three-dimensional array should be visualized as a two- or three-dimensional grid, not as a list of lists (or list of lists of lists), as it is now.

Each visualization method should be associated with a type. Taken to the extreme, this means embedding a complete 3D graphics package into our visual language.

If we view the constructs of our programming language as data themselves, then this might lead to a user-customizable syntax (which would presumably also require either meta-interpreter technology or reflective capabilities in the language).

Finally, we should strive to *apply the lessons we have learned to other programming paradigms.* Cube's computational model is based on higher-order Horn logic. We were attracted to this model because of its expressiveness and its (relatively) clean semantics. However, there are also problems associated with it, efficiency not being the least. We have learned that in the best case, the performance penalty one pays for a *visual syntax* is negligible. It would be interesting to devise a 3D notation for a language whose *semantics* is targeted towards run-time efficiency.

Bibliography

[1] James H. Andrews. Predicates as Parameters in Logic Programming: A Set-Theoretic Basis. In *Proceedings of Workshop on Extensions to Logic Programming*, Tübingen, Germany, December 1989. Published as *Lecture Notes in Artificial Intelligence*, 475:31 – 47, Springer-Verlag, 1989.

[2] Lennart Augustsson and Thomas Johnsson. Lazy ML user's manual. Chalmers University of Technology, Göteborg, Sweden, December 16, 1991.

[3] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[4] Marc H. Brown and James R. Meehan. The FormsVBT Reference Manual. Available via anonymous ftp from `gatekeeper.dec.com`, March 1993.

[5] Marc H. Brown and James R. Meehan. VBTkit Reference Manual — A toolkit for Trestle. Available via anonymous ftp from `gatekeeper.dec.com`, March 1993.

[6] Marc H. Brown and Marc A. Najork. Algorithm Animation using 3D Interactive Graphics. In *User Interface Software and Technology*, November 1993.

[7] Margaret M. Burnett. Abstraction in the demand-driven, temporal-assignment, visual language model. Ph. D. Thesis, University of Kansas, 1991.

[8] Margaret M. Burnett. Types and Type Inference in a Visual Programming Language. In *IEEE Symposium on Visual Languages*, pages 238 – 243, Bergen, Norway, 1993.

[9] M. Campanai, A. Del Bimbo, and P. Nesi. Using 3D Spatial Relationships for Image Retrieval by Contents. In *1992 IEEE Workshop on Visual Languages*, pages 184 – 190, Seattle, WA, September 1992.

[10] Luca Cardelli. Two-Dimensional Syntax for Functional Languages. In *Integrated Interactive Computing Systems*, pages 139 – 151, North-Holland Publishing Company, 1983.

[11] Shi-Kuo Chang, editor. *Visual Languages and Visual Programming.* Plenum Press, New York, 1990.

[12] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In Ewing L. Lusk and Ross A. Overbeck, editors, *Logic Programming: Proceedings of the North American Conference 1989*, pages 1090 – 1114, MIT Press, Cambridge, MA, 1989.

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

[14] Kenneth C. Cox and Gruia-Catalin Roman. Abstraction in algorithm animation. In *1992 IEEE Workshop on Visual Languages*, pages 18 – 24, Seattle, WA, September 1992.

[15] Carlos Christensen. An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language. In M. Klerer and J. Reinfelds (editors), *Interactive Systems for Experimental and Applied Mathematics*, Academic Press, New York, 1968.

[16] Carlos Christensen. An Introduction to AMBIT/L, and Diagrammatic Language for List Processing. In *Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, CA, 1971.

[17] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207 – 212, 1982.

[18] A. Del Bimbo, M.Campanai, and P. Nesi. 3-D Visual Query Language for Image Databases. *Journal of Visual Languages and Computing*, 3(3):257 – 271, September 1992.

[19] E. Denert, R. Franck, and W. Streng. PLAN2D – Towards a Two-Dimensional Programming Language. In *Gesellschaft für Informatik — 4. Jahrestagung*, Berlin, October 1974. Published as *Lecture Notes in Computer Science*, 26:202 – 213, Springer Verlag, 1974.

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

[21] T. O. Ellis, J. F. Heafner, and W. L. Sibley. *The GRAIL Project: An Experiment in Man-Machine Communications*, RAND Report RM-5999-ARPA, 1969.

[22] Anthony J. Field and Peter G. Harrison. *Functional Programming*. International Computer Science Series. Addison Wesley, 1988.

[23] George W. Furnas. New Graphical Reasoning Models for Understanding Graphical Interfaces. In *Proc. CHI '91*, pages 71 – 78, New Orleans, LA, 1991.

[24] Ephraim P. Glinert. Out of Flatland: Towards 3-D Visual Programming. In *Fall Joint Computer Conference*, pages 292 – 299, Dallas, TX, 1987.

[25] Ephraim P. Glinert and Steven L. Tanimoto. PICT: An Interactive, Graphical Programming Environment. *IEEE Computer* 17(11):7 – 25, November 1984.

[26] Warren D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225 – 230, 1981.

[27] F. Grant, L. McCarthy, M. Pontecorvo, and R. Stiles. Training in Virtual Environments. *Proceedings of the Intelligent Computer-Aided Training Conference*, Houston, TX, November 1991.

[28] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice-Hall, 1986.

[29] Daniel D. Hils. DataVis: A Visual Programming Language for Scientific Visualization. In *Proc. 1991 ACM Computer Science Conference*, pages 439 – 448, San Antonio, TX, March 5 – 7 1991.

[30] Daniel D. Hils. A Visual Programming Language for Visualization of Scientific Data. Ph.D. Thesis. Technical Report No. UIUCDCS-R-93-1809, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1993.

[31] C. M. Holt. *viz*: A visual language based on functions. In *1990 IEEE Workshop on Visual Languages*, pages 221 – 226, Skokie, IL, October 1990.

[32] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A Visual Programming Environment. *OOPSLA '88*, San Diego, September 1988. Appeared as *ACM Sigplan Notices*, 23(11):176 – 190, November 1988.

[33] International Organization for Standardisation (ISO). *Information Processing Systems – Computer Graphics – Graphics Kernel System (GKS)*. ISO 7942. ISO Central Secretatiat (1985).

[34] Joxan Jaffar, Spiro Michaylov, Peter J. Stuchey, and Roland H. C. Yap. The CLP($\mathcal{R}$) Language and System. Technical Report CMU-CS-90-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[35] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *1990 IEEE Workshop on Visual Languages*, pages 7 – 15, Skokie, IL, October 1990.

[36] Takayuki D. Kimura. Show and Tell Sample Programs. Technical report, Department of Computer Science, Washington University, St. Louis, MO, January 1986.

[37] Takayuki D. Kimura, Julie W. Choi, and Jane M. Mack. A visual language for keyboardless programming. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, MO, March 1986.

[38] Takayuki D. Kimura. Hyperflow: A Visual Programming Language for Pen Computers. In *IEEE Workshop on Visual Languages*, pages 125 – 132, Seattle, WA, 1992.

[39] Hideki Koike. An application of three-dimensional visualization to object-oriented programming. In *Advanced Visual Interface*, pages 180 – 192, Rome, Italy, 1992.

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

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

[42] Fred Lakin. Computing with text-graphic forms. In *Proceedings of the First Lisp Conference*, Stanford, 1980.

[43] T. K. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 202 – 217, Cambridge, MA, 1991. MIT Press.

[44] David Lau-Kee, Adam Billyard, Robin Faichney, Yasuo Kozata, Paul Otto, Mark Smith, and Ian Wilkinson. VPL: An active, Declarative Visual Programming System. In *1991 IEEE Workshop on Visual Languages*, pages 40 – 46, Kobe, Japan, October 1991.

[45] Henry Lieberman. A three-dimensional representation for program execution. In *1989 IEEE Workshop on Visual Languages*, pages 111 – 116, Rome, Italy, October 1989.

[46] John W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second edition, 1987.

[47] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. The Fabrik Programming Environment. *IEEE Workshop on Visual Languages*, pages 222 – 230, Pittsburgh, PA, 1988.

[48] Mark S. Manasse and Greg Nelson. Trestle Reference Manual. Technical Report 68, Digital Equipment Corp., Systems Research Center, Palo Alto, CA, December 1991.

[49] P. McLain and Takayuki D. Kimura. Show and Tell User's Manual. Technical Report WUCS-86-84, Department of Computer Science, Washington University, St. Louis, MO, March 1986.

[50] Dale A. Miller and Gopolan Nadathur. Higher-order logic programming. In *3rd International Conference on Logic Programming*, London. Published as *Lecture Notes in Computer Science*, 225:448 – 462, Springer-Verlag, New York, 1986.

[51] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348 – 375, 1978.

[52] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295 – 307, 1984.

[53] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *5th International Conference and Symposium on Logic Programming*, pages 810 – 827, 1988.

[54] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM Model for λProlog. In Ewing L. Lusk and Ross A. Overbeck, editors, *Logic Programming: Proceedings of the North American Conference 1989*, pages 1180 – 1198, MIT Press, Cambridge, MA, 1989.

[55] Marc A. Najork. Design and Implementation of the Cube Language — A Preliminary Examination Statement. University of Illinois, Department of Computer Science, March 1992.

[56] Marc A. Najork and Eric J. Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *IEEE Workshop on Visual Languages*, pages 215 – 220, Skokie, IL, October 1990.

[57] Marc A. Najork and Simon M. Kaplan. The Cube Language. In *IEEE Workshop on Visual Languages*, pages 218 – 224, Kobe, Japan, 1991.

[58] Marc A. Najork and Simon M. Kaplan. A Prototype Implementation of the Cube Language. In *IEEE Workshop on Visual Languages*, pages 270 – 272, Seattle, WA, 1992.

[59] Marc A. Najork and Simon M. Kaplan. Specifying Visual Languages with Conditional Set Rewrite Systems. In *IEEE Symposium on Visual Languages*, pages 12 – 18, Bergen, Norway, 1993.

[60] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[61] *apE: The Animation Production Environment – Version 1.1 Release Notes*. The Ohio Supercomputer Graphics Project, The Ohio Supercomputer Center, Columbus, OH, 1989.

[62] Rajeev K. Pandey and Margaret M. Burnett. Is it Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. In *IEEE Symposium on Visual Languages*, pages 344 – 351, Bergen, Norway, 1993.

[63] L. F. Pau and H. Olason. Visual Logic Programming. *Journal of Visual Languages and Computing*, 2(1):3 – 15, March 1991.

[64] Simon L. Peyton Jones. *The Implementation of Functional Languages.* Prentice Hall International Series in Computer Science. Prentice Hall, 1987.

[65] Jörg Poswig, Klaus Teves, Guido Vrankar, and Claudio Moraga. VisaVis – Contributions to Practice and Theory of Highly Interactive Visual Languages In *IEEE Workshop on Visual Languages*, pages 155 – 161, Seattle, WA, 1992.

[66] Jörg Poswig and Claudio Moraga. Incremental Type Systems and Implicit Parametric Overloading in Visual Languages. In *IEEE Symposium on Visual Languages*, pages 126 – 133, Bergen, Norway, 1993.

[67] Georg Raeder. A Survey of Current Graphical Programming Techniques. *IEEE Computer* 18(8):11 – 25, August 1985.

[68] Frank Van Reeth and Eddy Flerackers. Three-Dimensional Graphical Programming in CAEL. In *IEEE Symposium on Visual Languages*, pages 389 – 391, Bergen, Norway, 1993.

[69] Steven P. Reiss. A Framework for Abstract 3D Visualization. In *IEEE Symposium on Visual Languages*, pages 108 – 115, Bergen, Norway, 1993.

[70] Greg Rogers, Kelvin Sung, and William Kubitz. Combining Graphics and Windowing Standards in the XGKS System. *Computer Graphics Forum* 9:229 – 237, 1990.

[71] Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox, and Jerome Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, June 1992.

[72] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A Step towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431 – 446, 1990.

[73] Robert W. Scheifler and James Gettys. *X Window System.* Digital Press, third edition, 1992.

[74] Hikmet Senay and Santos G. Lazzeri. Graphical representation of logic programs and their behavior. In *1991 IEEE Workshop on Visual Languages*, pages 25 – 31, Kobe, Japan, October 1991.

[75] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold Company, New York, 1988.

[76] Silicon Graphics Computer Systems. *Iris Explorer User's Guide*. Document 007-1371-010, Silicon Graphics Computer Systems Inc., Mountain View, CA, 1992.

[77] David C. Smith. PYGMALION: A Creative Programming Environment. Ph.D. Thesis. Technical Report STAN-CS-75-499, Department of Computer Science, Stanford University, 1975.

[78] David C. Smith. *PYGMALION: A Computer Program to Model and Stimulate Creative Thought*. Birkhäuser Verlag, Basel, 1977.

[79] Lindsey Spratt and Allen Ambler. A Visual Logic Programming Language Based on Sets and Partitioning Contraints. In *IEEE Symposium on Visual Languages*, pages 204 – 208, Bergen, Norway, 1993.

[80] John T. Stasko and Joseph F. Wehrli. Three-Dimensional Computation Visualization. In *IEEE Symposium on Visual Languages*, pages 100 – 107, Bergen, Norway, 1993.

[81] Randy Stiles and Michael Pontecorvo. Lingua Graphica: A visual language for virtual environments. In *1992 IEEE Workshop on Visual Languages*, pages 225 – 227, Seattle, WA, September 1992.

[82] Ivan E. Sutherland. Sketchpad, A Man-Machine Graphical Communication System. In *Proceedings of the AFIPS Spring Joint Computer Conference* 23:329 – 346, 1963.

[83] Ivan E. Sutherland. The Ultimate Display. In *Proceedings of the IFIP Congress*, pages 506 – 508, 1965.

[84] William R. Sutherland. *On-Line Graphical Specifications of Computer Procedures*. Ph.D. thesis, MIT, Cambridge, MA, 1966.

[85] Ivan E. Sutherland. A Head-Mounted Three-Dimensional Display. In *Proceedings of the AFIPS Spring Joint Computer Conference* 33:757 – 764, 1968.

[86] David A. Turner. MIRANDA — a non-strict functional language with polymorphic types. In *Conference on Functional Programming and Computer Architecture*, Nancy, France, 1985. Published as *Lecture Notes in Computer Science*, 201:1 – 16, Springer-Verlag, New York, 1985.

[87] C. Upson, T. Faulhaber Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30 – 42, July 1989

[88] Konrad Zuse. *Beschreibung des Plankalküls.* R. Oldenbourg, München, 1977.

Vita

Marc Alexander Najork was born on August 14, 1963, in Frankfurt, Germany. He attended Geschwister-Scholl Schule in Schwalbach and Altkönigschule in Kronberg, from which he received a High School degree in December 1981. Between April 1982 and June 1983, Marc completed the mandatory service in the German Army.

In October 1983, he entered the Technical University of Darmstadt to study *Wirtschaftsinformatik*, a program which merges Computer Science with Business Administration, Economics, and Law.

While studying at Darmstadt, he received a scholarship from the German Academic Exchange Service which enabled him to spend a year abroad. He spent the period from August 1987 to May 1988 as a non-degree student in the College of Engineering of the University of Illinois at Urbana-Champaign.

Marc received the degree of Diplom-Wirtschaftsinformatiker from the Technical University of Darmstadt in May 1989.

In January 1989, Marc joined the Department of Computer Science at the University of Illinois to pursue his graduate education. His adviser was Prof. Simon Kaplan. During this time, he held positions as Teaching and as Research Assistant. He also was awarded two University Fellowships.

After receiving his Ph.D., Dr. Najork joined Digital Equipment Corporation. He is currently a Principal Software Engineer at the Systems Research Center in Palo Alto, California.