Diss. ETH No. 11817

Wolfgang Weck

# On Document-Centered Mathematical Component Software

1996

# On Document-Centered Mathematical Component Software

# On Document-Centered Mathematical Component Software

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich (ETH Zürich)
for the degree of
Doctor of Technical Sciences
presented by

Wolfgang Weck
Dipl. Informatik-Ing. ETH
born December 7, 1964
citizen of Germany

accepted on the recommendation of
Prof. G. H. Gonnet, examiner
Prof. N. Wirth, co-examiner

Diss. ETH No. 11817
1996

# Acknowledgements

This work would not have been possible without the support of many people. I am most indebted to Prof. Gaston Gonnet for his supervision. His competence in computer algebra systems provided an important background for me. I also want to thank Prof. Niklaus Wirth for acting as my co-examiner. I have to admire his way of concentrating on the essence in system design.

It is a pleasure to mention all my colleagues from the Institutes of Scientific Computing and of Computer Systems. Through all the years I enjoyed many fruitful discussions about various aspects of computer science in particular with Dominik Gruntz, Lukas Knecht, Stefan Ludwig, Cuno Pfister, and Clemens Szyperski. I also want to thank Stefan Ludwig, Clemens Szyperski, and Stefan Vorkötter for proofreading many of my earlier writings.

My work led to several student projects also. Rolf Eberhardt used my ideas to build a document-centered user interface for his music searching system. Walter Schmidli implemented a linear algebra package and helped in developing a first scripting mechanism. Martin Roth implemented many drawing routines for mathematical expressions. The enthusiasm of all three turned our cooperation into a pleasure.

Last but not least, many thanks to the Institute of Computer Systems for two of their precious Ceres computers.

# Contents

# Zusammenfassung

Es ist erstrebenswert, Software aus Komponenten zusammenzustellen, die unabhängig voneinander hergestellt und vermarktet werden. Solche Komponenten müssen innerhalb eines Frameworks entwickelt werden, um später kombiniert werden zu können. Dieses Framework hängt vom Anwendungsgebiet der Software ab.

Parallel zur komponentenorientierten Softwarearchitektur benötigt man ein erweiterbares Benutzerinterface. Dokumente sind eine gute Basis dafür. Einzelne Softwarekomponenten werden als Editoren für Dokumente oder für Teile davon präsentiert. Solche Software wird als dokumentenzentriert bezeichnet.

In dieser Dissertation wird ein Framework für mathematische Komponentensoftware präsentiert und ein dokumentenzentriertes Benutzungsmodel entwickelt. Beide sind eine Fallstudie für das Design dokumentenzentrierter Komponentensoftware im allgemeinen, und zugleich ein konkreter Vorschlag im Bereich mathematischer Software.

Durch die Benutzung einer allgemeinen Programmierumgebung anstelle eines Systems für mathematische Software, können existierende Dokumenteneditoren weiter benutzt werden. Die Möglichkeit zur Kombination mit weiteren Komponenten, z.B. für Hypertext, ist ein wichtiger Schritt in Richtung interaktiver Lehrbücher.

Die wesentlichen Beiträge der Dissertation sind folgende:

Mathematische Software sollte in drei Richtungen erweiterbar sein: Typen mathematischer Ausdrücke, Algorithmen und Editoren. In unserer Architektur werden die mathematischen Ausdrücke zu Verbindungselementen zwischen berechnender Software und den Editoren. Mathematische Ausdrücke in Dokumenten dürfen nicht durch Automatismen verändert

werden. Es darf beispielsweise keine Normalform erzwungen werden, wie dies viele auf das Rechnen gerichtete Systeme tun. Da gleichzeitig dieselben Ausdrücke für Berechnungen benutzt werden sollen, müssen sie mathematische Objekte repräsentieren, nicht nur Grafiken. Daher ist eine Abstraktion mathematischer Ausdrücke, die beiden Aspekten gerecht wird, fundamental in dieser Arbeit.

In erweiterbarer mathematischer Komponentensoftware müssen Ausdrücke als unveränderbare, gerichtete, azyklische Graphen dargestellt werden. Es wird ein Design Pattern für solche Graphen eingeführt. Dieses erzwingt die nötigen Restriktionen, ist zugleich aber bequem zu benutzen.

Compound-Documents können als Datenspeicher benutzt werden. Sie haben einige Vorteile gegenüber solchen, die auf Zuweisungen an Namen beruhen. Spezielle Dokumenten-Parts repräsentieren mathematische Ausdrücke. Eine neue Idee ist, solche Dokumenten-Parts auch zu benutzen um zukünftige Resultate zu repräsentieren, die parallel berechnet werden. Ein weiteres neues Konzept ist Dokumenten-Parts in ihrem Kontext zu interpretieren. Damit lässt sich ein einfacher Formeleditor und ein Interpreter für Skripte herstellen.

# Abstract

The idea of composing software from components, which are created and marketed independently of each other is appealing. To be composable later, such components need to be developed within some framework. This framework depends on the software's domain of application.

In parallel to the component-oriented software architecture an extensible user interface is needed. Documents are a good basis for this. Individual software components are presented as editors of documents or parts thereof. Such software is called document-centered.

In this thesis a framework for mathematical component software is presented and a document-centered user model is developed. Both are a case-study in the design of document-centered component software in general and, at the same time, a concrete suggestion in the domain of mathematical software.

Using a general purpose programming environment instead of a mathematical computation system allows us to reuse existing document editors. Composability with further facilities, like hypertext support, is an important step towards interactive textbooks.

The main contributions of the thesis are the following:

Mathematical software should be extensible in three directions: expression types, algorithms, and editors. In our architecture, expressions are used to link computational software and editors. Expressions within documents must not be changed automatically. For instance, the software must not enforce a normal form, like many systems focused on computing would do. Since at the same time, the same expressions shall be used for calculating, they must represent mathematical objects, not just graphics.

Therefore, an abstraction of expressions, allowing to deal with both issues, is fundamental to this work.

With extensible mathematical component software, expressions must be stored as immutable, directed, acyclic graphs. We introduce a new design pattern for such graphs. This design pattern enforces the necessary restrictions, yet clients can use it conveniently.

Compound documents can be used as data repositories, having certain advantages over such based on name bindings. Special document parts represent expressions. A new idea is to use such document parts also to represent upcoming results, being computed concurrently. Another new concept is to interpret document parts within their context. This is used to create a simple expression editor and a script interpreter.

# 1 Introduction

*It is vain to do with more what can be done with less.*
*("Occam's Razor", William of Occam 1285–1347)*

## 1.1 Why Mathematical Component Software?

*Mathematical software* includes software for symbolic and numeric computing as well as for expression editing and typesetting. Nowadays, computing software is mostly presented as application programs, so-called "systems". Prominent examples of such systems are Maple [CGGLMW–91], Mathematica [Wolfram–88], Axiom [JeSu–88], and Matlab [Mat–90]. They can be extended through programming in a proprietary language. Consequently, each of the systems implements its own user interface, expression editor, pretty printer, and language interpreter instead of striving for being composable with external components. According to [Wirth–95], this software has to be considered as "fat".

Also, to solve a particular problem, one wants to combine the best tools. With the application program approach either each program has to come with the best algorithms for every possible problem or the data to operate on must be transferred between programs. It has already been recognized that the former approach is not suitable, and steps have been made towards the latter, e.g. in [Kajler–92a, OM–96]. These attempts focus on reuse of existing software, thus making the entire systems not leaner but fatter.

This thesis shows that lean mathematical component software is possible. Small collections of mathematical algorithms can be made available as individual components. They can be combined by the user as needed. They also allow for bidirectional integration with other, already existing software components. Other software within the same component software environment could use computing components, without being forced to import a complete computer algebra system.

Nowadays, the need for such integration becomes more and more apparent. With today's ideas about interactive textbooks it becomes important to have access to document organizing tools, like hypertext, outline editors, etc. As a consequence, fat computer algebra systems become even fatter with more and more typesetting and hypertext facilities. An example of this development is Mathematica's user interface, presented in [Soiffer–95]. The same tools are built for every system, and they will never cooperate. This situation can be overcome by the component software approach.

## 1.2 Why Document-Centered Presentation?

Nobody wants to start up a symbolic computation system. What people want is the computer to operate on their data. In many cases, this data will be represented within some kind of document. Such a document can be an e-mail message received from a colleague, a scientific paper retrieved from the world-wide web, or even an interactive text book. In this context, computing is viewed as equivalent to edit these documents, i.e. to change their contents.

If the user's activity is centered around editing documents, the software is called *document-centered*. In contrast, *process-centered* software organizes the user's activity as several sequential threads.

[ABMW–88] and [Sydow–92] state that mathematical software should not be encapsulated in big, process-centered, application programs but be presented as document editing operations.

Additional support for document-centered software comes from component-oriented programming: extensible software needs extensible user interfaces. With OpenDoc [OD–94] and OLE [Brockschmidt–94], compound documents are the platform on which software components can build their user interface.

## 1.3 What Is The Problem?

The goal is to make mathematical software available in the form of small, independently developed software components, which can be combined by the end user. Mathematical software components shall be combinable with each other, but also with non-mathematical components, like document processing software.

This main goal presents two problems. First, an architecture for mathematical component software has to be found. Today's mathematical software does not have the required properties, it is not structured into reusable components. Secondly, a document-centered software presentation must be developed. This requires to construct a new user model.

An all purpose programming environment must be useds as environment for mathematical software components. Only this allows to combine the new mathematical software components with other, possibly already existing, software. The one and only task of such an environment is to provide the fundamentals for component-oriented programming. Within this environment another framework is to be created. It must implement the fundamental domain-specific abstractions, elsewhere designed into a special purpose language. Examples for such abstractions are an abstract data type for mathematical expressions or interfaces of components that provide certain services, like printing or accessing selections made by the user.

As stated by Deutsch [Deutsch–89] and others, designing these abstractions and interfaces is difficult. On one hand, these abstractions are to be used by every extension programmer. Power and ease of use are important. On the other hand, the abstractions need to be implemented for every extension. Therefore, they should be as simple as possible. Finally, these abstractions also serve to protect the extension components from interference. Hence, their correct design is a major safety issue.

The second difficulty is to model the abstractions found with the tools at hand. Abstractions are useful only if the invariants they rely on can be asserted by the environment. In open, extensible systems only interface specifications can be used to enforce global invariants. Designing interfaces to assert the required invariants is a major challenge.

Also, some problems are to be solved on the document-based user interface level. Allowing to insert mathematical expressions into a document is not enough. Support for scripting and for concurrent computation (e.g. on a remote server) is a must for an interface to computer algebra. The latter raises the problem of how concurrent operations, requiring long computation time, can be reasonably integrated into an interactive document editor.

The documents shall also form a foundation for interactive textbooks. In this context, the design of the data structures to represent mathematical expressions is particularly difficult. The data structure must represent enough information for presentation of expressions within documents, and it must be usable by computation software. In contrast to what present computer algebra systems do, the presentation of an expression must not be changed.

Mathematically correct simplifications (like removing parentheses) must not be enforced. Otherwise, the following equation could not be presented within a document:

$$(a+b)+c=a+(b+c)$$

## 1.4 What Should Be Reused?

Since the advent of object-oriented programming the term *reuse* has become a frequently used buzzword. Unfortunately, sometimes it is understood as the allowance to use any existing implementation of software ad infinitum.

It is an old observation, stated for instance in [Johnson–88, Deutsch–89, Magnusson–91], that software is extensible and reusable only when designed with this purpose in mind. Consequently, the framework for mathematical component software presented in this thesis is a new design. It allows for clearly specified extensibility.

During the migration phase existing software could be linked with the new framework. The client/server model can be used for this. Therefore the implementation, underlying this thesis, features a link to Maple. Still, the driving force behind the research is to create lean software, not to minimize the work necessary to switch to it. "Doing with less" in Occam's razor means that the resulting software demands less resources, not that the design phase requires less effort, nor to avoid (systematic) redesign of existing software.

It must also be said, that using existing extensible abstractions can be a pleasure. This thesis presents a light weight implementation of a central, directly manipulatable mathematical object repository based on compound documents, as an example.

## 1.5 Road Map

The rest of this thesis supports the statements and claims made above. To that end, a framework for mathematical component software is outlined. On this framework and a compound document framework a particular light weight extension implements a document-centered user interface. An implementation is presented, that serves as proof that the entire design can be implemented and used.

**Chapter 2** introduces the fundamental techniques and terminology used throughout the thesis. Fundamental properties of interactive application programs are presented together with their impact on software architecture, extensibility, composability, and user interfaces. This shows the limits of the process-centered approach. As a contrast, component-oriented programming and independently extensible systems are introduced. Their demands are discussed to derive the main requirements for component framework design. Also, a brief introduction is given to compound documents as the base for document-centered software.

**Chapter 3** suggests an architecture for mathematical component software. This architecture is reflected by a component framework. The latter allows for independent extension with expression types, computing algorithms, and editing facilities. It offers central services and asserts system-wide invariants.

**Chapter 4** elaborates on the document-centered software presentation. In the overall architecture it assumes the role of an extension implementing the user interface, including expression editing. First, we show how workspaces – the user's object stores – can be implemented by compound documents. A generalized notion for icons is defined. Computational operations are turned into or viewed as editing operations. Secondly, scripting is addressed. It is shown how compound documents can be employed here. Thirdly, an expression editor is described. It is based on compound text documents and icons, but also allowing for graphical editing. Very flexible palettes with templates and overlays can be offered. Most importantly, the editor is generic, i.e. independent of expression types.

**Chapter 5** shows that the component framework suggested in Chapter 3 can be implemented, that it can be extended as claimed, and that it implements sufficient functionality. The essential parts of an existing implementation of the framework itself and of some extensions are shown as proof. The environment is the Oberon System [WiGu–89] together with the programming language Oberon-2 [Mössenböck–91].

**Chapter 6** discusses related work. Approaches to improve the software structure other than independent extensibility are reviewed. Furthermore, existing user interfaces of mathematical software, existing expression editors, and typesetting implementations are discussed.

**Chapter 7** presents some conclusions.

# 2 Fundamentals And Terminology

This chapter briefly reviews concepts fundamental to this thesis. First, the impact of presenting software as interactive application programs is considered. Most of today's mathematical software has this structure. In particular, the notion of workspaces as a program's data repository is introduced.

Secondly, as an alternative, component-oriented software is discussed. In contrast to object-orientation, assembling systems from prefabricated components can be left to the user. The underlying component frameworks have to be independently extensible. This is the main goal to be met by the design of the mathematical component framework, shown in Chapter 3.

Finally, extensible systems require extensible user interfaces. These are based on compound documents. As a result, the software presentation becomes document-centered, rather than application program centered.

## 2.1 Interactive Application Programs

Traditional computer systems are extended with new functionality by *application programs* (or *programs* for short), which can be loaded into the system and executed. Some programs will terminate and be removed from the system immediately after having performed their task. Others will regularly accept input from the user, directing their further execution. These latter programs are called *interactive application programs*. Today's systems allow to run several interactive programs simultaneously, but restrict interaction between them heavily. This has an impact on the overall structure of the software and its user interaction model.

### 2.1.1 Interactive Application Programs Are Encapsulation Units

Interactive application programs are, in general, a package deal. Apart from being the mechanism to extend the system's functionality, they encapsulate functionality, data, and user interface aspects.

#### Encapsulation Of User Interfaces

Many user interfaces to interactive application programs are built on sequential streams. The user has to connect a terminal to such a stream and to conduct a dialog with the program. Such terminals can be windows within a multi-window environment. Nowadays, the typewriter style of these interfaces is being replaced by graphical representations, like Maple *Worksheets* [CGGLMW–91] or Mathematica *Notebooks* [Soiffer–95]. Many programs for mathematics can also display several windows at the same time, e.g. display plots in separate windows.

Still, each window belongs to exactly one instance of an application program. When a program terminates, all its windows vanish.

Every program carries its own, individual user interface functionality. Each user interface follows its own model and conventions, requiring the user to learn them.

#### Encapsulation Of Data

While using an interactive application program, the user manipulates data which needs to be stored and managed within the computer. The data of different programs is markedly separated. The intention of this storage protection is to limit accidental data destruction to single programs. This is necessary only, if the programming environment lacks memory protection on a finer granularity, e.g. on the basis of individual objects, or information hiding on component level.

There is no way for programs to share data. Multi-window environments try to make up for this by supporting copy operations between windows, but this can be seen only as a tool covering communication between two separated programs. The data has to be replicated in the memory of the target program.

#### Encapsulation Of Functionality

The main purpose of an interactive application program is to add new functionality to the system. The user can access this functionality through communication with the program. This communication based interface is not suitable for software reuse. Without a programming interface, a user must be simulated to make use of a program's functionality within another program.

Also, programming interfaces provided by such interactive systems are often very restricted. The one offered by Maple, for instance, basically allows to execute the same kind of operations that are available to the user.

It is due to these encapsulation properties that application programs are not truly open. They can be combined with each other as monoliths only, not as components contributing small sets of functionality. Furthermore, to apply functionality of one program to data stored within another one, the data needs to be replicated.

Because of this, it is not possible to get a part of a system alone. Finally, extending programs with new functionality is impossible in principle. The only exception are programs that implement programming environments on their own. These extension mechanisms are proprietary and thus do not support interaction of different programs (see also Section 6.2).

### 2.1.2 Workspaces

Interactive application programs have to manage the user's data. The data is represented as part of the system's internal state and the user will change that state, step by step. This state is called the user's *workspace*.

This thesis deals with mathematical data (expressions), but the workspace concept is more general. Data within a workspace can also represent the result of a data base query, for instance. It is a characteristic of the workspace implementation through documents (described in Section 4.1) that a single workspace can contain arbitrary types of data objects simultaneously.

The term *workspace* has been used several times, for instance with APL [GiRo–74], Axiom [JeSu–88], and Matlab [Mat–90]. It appears that it has not yet been defined precisely. (Only in the glossary of [JeSu–88] a workspace is defined as the record of an interactive session.)

In this thesis a workspace is understood, like with APL, to be the data repository accessed by both the user and the computing software. Manipulation of a workspace requires some specific functionality. From this, the following definition can be derived.

**Definition:** A *workspace* is a repository for data objects generated and manipulated by the user of interactive software. It is to be accessed by both the user and the software. Its specific features are to allow for

- insertion, replacement, and removal of objects
- an overview over the objects currently being in the workspace
- reference to objects as input for operations
- visualization of objects
- externalization and internalization of the state to and from files.

Some computer algebra systems like Maple [CGGLMW–91] or Mathematica [Wolfram–88] maintain data repositories at two levels. On one hand the user can store objects within documents maintained by the user interface. On the other hand the system's kernel maintains an object repository, using name bindings as references.

The latter is the workspace of the system. Only objects in the system's repository can be used for computation. Data stored in the user interface must be transferred to the internal workspace before computing with it. This transfer may be implicit upon invocation of a computation.

Regardless of what the document maintained by the user interface shows, the result of a computation depends on the name-bindings made by the system's kernel. If the computation involves a symbol $x$, the outcome depends on whether an object is bound to $x$, or not. Synchronization between the user interface's workspace and the system's workspace is a major problem.

### 2.1.3 Workspaces Implemented Through Name Bindings

Interactive application programs allow the user to conduct a dialog with the program while it is active. This dialog is based on a language interpreter. The language can usually be used for both directing the system and programming. Inspired by the latter, the workspace management is based on name bindings.

The concept of names is taken with generality here. A name can also be a number (in system's numbering all their input and output expressions, e.g. Mathematica) or a sequence of special characters (like those used by Maple to denote recent results).

At a certain point in time the workspace contains exactly those objects which are currently bound to at least one name. Insertion, replacement, and removal are mapped to the corresponding assignment actions. These can be explicit (assignment commands issued by the user) or implicit (side effects of computational commands). Objects can be referred to by their name. Usually, to provide an overview all names in use at a specific moment can be listed through a command. Other commands allow to store the contents of the workspace to a file or read it from a file.

The implementation of workspaces using name bindings has one disadvantage. The name bindings are not visible directly. The user can inspect this hidden state through special commands only. Such commands require extra activity of the user and create only snapshots of the state, whereas the state is transformed permanently during a session. Therefore, to work effectively with such a system the user must keep a mental picture of the hidden state. Assignments need to be remembered to make use of them. The particular danger is that a data object is lost by accidental reuse of the name it had been bound to or temporarily lost by the user forgetting the name he/she has used.

In the realm of programming, name bindings present no problems. This is due to the static scoping allowing for proofs of predicates about variables during their entire life time. In an interactive program the session creates the only scope for the bindings. This scope is not static but a temporal instance (dynamic scoping). No static review or derivation of properties of variables is possible.

### 2.1.4 Sessions, History And Hidden State

The entire dialog between the user and the program is called a *session*. A session begins when the program is started and lasts until the program's termination. Usually it is presented to the user by a window or a terminal. In name binding based implementations of workspaces the session also creates the scope for the bindings. Note, that in contrast to programming this scope is not defined by a static construct like a procedure or a module (lexical scoping), but by a temporal instance (dynamic scoping).

We call the sequence of commands executed during a session up to a certain point of time the *history* of this session. It is of particular importance whenever objects in the workspace are referred to using *hidden state*. A table of bindings between names and objects is such a hidden state. Because the state will be transformed during a session, the history of this transformations is essential to understand the current state. In other words: the user has to remember previous assignments (or to review them in a protocol) to be able to use them.

Starting, stopping, and interrupting of programs and commands is understood as *session control*. Most of this is an artifact of the interactive application program paradigm.

## 2.2 Component Software

### 2.2.1 What Is Component Software?

*Component software* is a software structuring principle based on combination of software components. *Software components*, in turn, are units of extension, encapsulation, and abstraction with a granularity coarser than objects but finer than programs. Components can be used as prefabricated units to compose higher order systems, but they can also be made available independently to be combined by the user.

#### Components are larger than objects

Normally, components bundle several objects. Therefore, they offer more powerful abstractions than individual objects. In particular, a component hides the interaction of various objects. This has two advantages: on one hand, the client of a component is not left alone with a bag of objects and classes difficult to survey. Instead, a more powerful abstraction, also covering relationships between objects, is created. On the other hand, information hiding can be employed on a higher level: Szyperski [Szyperski–92a] shows the importance of enforcing invariants that involve more than one object within a component. This makes components more secure. An example are the components of the Oberon system implementing Carrier-Rider pairs. This has been used to implement immutable objects in an efficient way (see Section 5.1).

#### Components are smaller than programs

The important property of component software is the possibility to combine components on demand and to extend even a running system with new components. Functionality is added to the system, when it is needed for the first time. Usually, interactive application programs are not extensible, once they are loaded and launched. (The exception of extensible application programs is discussed in Section 6.2) Thus, programs try to cover a maximum of functionality, whereas components try to encapsulate a minimum, which makes sense as a building block for a larger system.

This principle was already used in the UNIX system. Simple UNIX commands, that transform byte streams, can easily be combined through redirecting the input and output streams. However, the fact that the interfaces between such "components" are limited to byte streams is a severe limitation. Most importantly, no type checking across interfaces is possible.

The finer granularity of components is feasible only because of expressive programming interfaces. The security of statically type-checked programming interfaces is a prerequisite in any open system (see Section 2.2.4). Simultaneously, structured typed interfaces document themselves implicitly. When application programs are connected via streams the interface description is less expressive – it features the sequence of bytes as its only static type. The semantics of a byte stream depends on its interpretation. It cannot be asserted that both sender and receiver interpret the bytes consistently. Also, there is an extra cost in run time and software complexity to translate between sequential and internal representation of data.

Application programs are usually associated with heavy weight processes, each having separate protected memory. Components use one common address space, they share memory. Objects can be passed between components by reference and need not be replicated. Connecting programs is more difficult than connecting components. Therefore, with respect to the suitability as building block, components are smaller than programs.

Component software separates data objects from software components at the top of the abstraction hierarchy. Data structures are organized completely independently of the component hierarchy. In fact, many data structures are inhomogeneous and even generic, allowing new components to define new object types and have them included in system wide structures.

Data is shared among components. Memory protection must be done for individual objects rather than for entire programs or components.

### 2.2.2 Independently Extensible Systems

To realize the full potential of component software the user must be enabled to choose and combine components upon individual needs. There will be multiple suppliers who provide components independent of each other. The ultimate goal for component software systems therefore is to support independent extensibility.

**Definition:** A software system is called *independently extensible* if it can be extended with new components at any time, and if those components can be designed by independent suppliers in complete ignorance of each other.

System integration (composition) is done by the customer at run time. Programmers have to obey to certain rules to avoid that components mutually interfere. Exceptions to the latter are components that perform mutual exclusive tasks or require global resources that cannot be shared, e.g. the serial communication port. However, as stated in [Szyperski–96], such mutual exclusion must be apparent to the user.

Independent extensibility is a prerequisite for lean software as defined by [WiGu–89, Wirth–95].

From an industrial point of view, the purpose of independent extensibility is twofold. On one hand, the customer gets exactly the functionality demanded by combining suitable components. On the other hand, a component software market allows for more competition. Customers can buy and combine components from different vendors. Components can be selected based on their individual ratio of price and performance (functionality, efficiency, quality). According to Beech [Beech–96], producers will be able to concentrate on their particular strengths and deliver specialized components of high quality without being forced to build complete systems. This concept already works on the hardware market, where extension mechanisms like SCSI allow to compose hardware components of independent suppliers. A software component market, however, will need a wider spectrum of application domain specific interface standards.

Components developed by different vendors in complete ignorance of each other must not interfere. Of course, some components will be mutually exclusive, like those using a global hardware resource (the serial communication port of the machine, for instance). Such situations are typically transparent to the customer and can be resolved at configuration time.

In many cases cooperation between independently created components is required. This needs some standardization for information exchange.

### 2.2.3 Independently Extensible Component Frameworks

Design patterns are solution recipes, applicable to families of problems. *Frameworks* are implementations of design patterns as defined by [GHJV–95]. Both consist of several classes and objects, defining their relations. Frameworks are extensible, allowing specialization for particular problems. If the result is a complete application, the framework is called an *application framework*.

*Component frameworks* are extensible with components. They do not primarily support specialization of solutions, but they offer basic services for extension components. For instance, they define interfaces for information exchange and cooperation of components. They also define aspects of the architecture of such components. These depend on the classes and extension mechanisms of the framework. Therefore, component frameworks implicitly include design patterns for the overall component system.

Safety considerations are of particular importance in extensible systems. Misbehavior of a single component must not corrupt the system as a whole. This aspect is usually covered by designing frameworks such that they protect their own data and resources.

With independent extensibility, one must go a step further. Extension components must get along with each other but not into each other's way, even though their respective designers may not know about each other. Therefore, component designers will have to follow several rules. Such rules can include system-wide invariants. They depend on the particular purpose, domain, and architecture of a system.

In the past, this has often been approached by sets of documented guidelines. Examples are naming conventions to avoid name clashes in global scopes. It would be preferable, however, if such conventions were not needed. The alternative are abstractions and assertions made by the underlying framework.

In [Szyperski–96] it is stated that independently extensible systems do not allow for any global, static analysis, since they are never complete. The only tool to enforce system-wide invariants statically is the interface specification of components. The compiler can assert correct implementation and correct usage of such a specification. To express system-wide properties within an interface specification, information hiding and static typing must be used.

As a consequence, interface design of independently extensible component frameworks is the ultimate challenge for a contemporary software engineer. The correct rules for extension providers must be identified, and they must be modeled with the tools at hand. Still, restrictions must not hinder extensibility and usage more than necessary.

A typical example, where many of today's programming environments fail to support safe independent extensibility, is the lack of separated name scopes for extensions. In a single global scope, the only way to prevent name clashes is to stick to a convention (and to hope that everybody else does so) like the one mentioned above. Modular programming languages like Modula-2 [Wirth–82], Oberon [Wirth–88], or ADA [Ada–83] solve this problem in a better way separating the name scopes of the modules. Then, only the module names must be chosen uniquely. Since in most such environments module names correspond to file names, module name clashes are likely to be detected statically, before the system is started.

### 2.2.4 Requirements To Component Software Environments

Component software environments are component frameworks themselves. They may be extended by complete components but also by further component frameworks, specialized for specific domains. Four major requirements for a component software environment can be identified:

#### Programming Interface Specification

To allow components to interact with each other it is necessary to agree on a system wide description of programming interfaces. Such a description allows for static checks that interfaces are implemented and used correctly, similar to separate compilation with interface checking as used with Modula-2 and Oberon (in contrast to independent compilation without such checking) and described for instance in [Crelier–94].

At a first glance, assertions in an interface seem to contradict the requirement of extensibility. To solve this dilemma the interface specification formalism needs to allow to express extensibility. The Oberon programming language therefore features type extension, which is the most important difference to its predecessor Modula-2.

#### Dynamic Loading

Components must be loaded and linked into the system dynamically, i.e. at run time. This requires a loading linker with a programming interface. The user should not be required to explicitly load a component, before being able to use it. Rather, a component is loaded automatically, when a command is issued that requires it, or when data is loaded which is defined in it. The later occurs for instance when a compound document is opened. Linking loaders are available in several industrial operating systems.

#### Automatic Memory Retrieval

Since data is shared between components using references, no component has exclusive control over such data. Once a reference has crossed the barrier between two components none of them can decide how long the object is accessible and when the memory can be retrieved. Therefore, automatic storage reclamation (garbage collection) is a necessity.

#### Open User Interface

It is not sufficient that run time extensions are supported by the software system. Extensions must also be accessible to the user; an extensible user interface is required. Traditionally, the environment offers a window system as the integration base. Application programs extend the user interface with new windows. As application programs are markedly separated in memory, they are in the user interface. Finer grained integration of components needs finer grained user interface integration. This is offered by the compound document concept. It allows not only to have different windows on one screen, but to put individual objects into a variety of documents. Consequently, from compound document frameworks more is required than from window systems. The latter abstract bitmaps, but the former need to abstract entire documents, related resources (like files), and editing means.

### 2.2.5 Existing Component Software And Open Document Environments

An environment for component software needs to offer more than one for application programs. Component software requires to express and check interfaces statically, to manage dynamic loading and binding, to automatically retrieve free memory, and to define a user interface framework. Application programs in turn interface each other by primitive byte streams only, are loaded separate of each other once before launched, manage their own memory and implement their own user interface. Thus, traditional operating systems are not usable as component software frameworks.

Current software industry begins migration from traditional application programs to component software. Right now, two competing standards for future component software environments exist: OpenDoc [OD–94] and OLE [Brockschmidt–94]. Independent of these the Oberon system has grown mature. From the beginning, it had been designed with extensibility in mind.

#### Visual Basic

Visual Basic is not a true component software framework. It allows to combine various existing components using scripting, but does not include support for component programming.

#### Delphi

Delphi is based on an object-oriented version of Pascal. It can be extended with self-programmed components. To be a real component framework, it lacks garbage collection.

#### SOM/OpenDoc

IBM defined the interface standard SOM (System Object Model). It uses a new language to express interface properties. This makes interface definitions independent of the language used for implementation but also makes it more difficult to assert correct interface implementation and usage. For the programmer it makes keeping interface and implementation consistent more difficult.

Dynamic loading is contributed by the underlying operating system. Automatic memory retrieval is not supported by SOM. This is probably its weakest point. OpenDoc is the accompanying standard for compound documents. It provides integration means for SOM objects.

### COM/OLE

The Component Object Model (COM) has been defined by Microsoft. It competes with SOM to which it is comparable with respect to potential and general approach. The most important single difference is that a reference counting scheme to support memory retrieval has been included into COM. The respective compound document standard is defined in OLE [Brockschmidt–94].

### Oberon

During the last ten years, the Oberon system [WiGu–89, WiGu–92] has become mature. Inspired by Cedar [Teitelman–84], it had been designed as an extensible system from the very beginning. Since Oberon has been used to implement the expression framework discussed in this thesis, a more detailed description is given in the following.

One of Oberon's most distinctive features is the strong integration of the environment and the programming language which is also called Oberon [Wirth–88, Mössenböck–91]. The latter had been derived from Modula-2 [Wirth–82] and enhanced with type extension.

Components are modules. The compiler checks correctness of both usage and implementation of module interfaces. The symbol files used for this purpose are generated automatically during compilation of a module. Modules are loaded and linked into the system dynamically. The common memory is managed with the help of a mark-and-sweep garbage collector.

At the user interface level extension is possible through new viewers and commands. Viewers are windows which make coarse grained objects (entire texts, graphics, pictures, etc.) accessible to the user and allow for their manipulation through command execution. Commands are procedures without parameters. They operate on the system's global state. They are activated either through direct manipulation operations within a viewer, or by clicking the mouse on a command's name in a text. The latter gives the user large flexibility, since menu configuration etc. become obsolete. Also, there is no need for a global menu bar on the screen which is being changed, depending on the current focus. The possibility to include commands and their parameters into documents, makes Oberon a hypertext environment.

In the meantime various extensions of the core system have been introduced. Two extensions target at open documents: Oberon System-3 [Gutknecht–94] has a platform for graphical user interfaces. These are composed from objects which can be bundled in various containers, like two-dimensional panels or texts. Behind the scene, the introduction of a global base type from which any object is to be derived and the support of object libraries are the most remarkable change.

Oberon V4 evolved from the original system with less fundamental changes: it differs mainly in the text model described in [Szyperski–92], which was inspired from the Ethos Project [Szyperski–92b]. A text is now defined as a sequence of objects attributed with font, size, color and vertical offset. Such objects are mostly plain characters, represented by their ASCII code. An abstract class, called *Text Element* (or *Elem* for short), can be extended to embed any self designed object into texts. With this implementation compound documents are restricted to texts. This restriction is somewhat mitigated by the fact, that text can be embedded in many other models.

Recently, a framework, called Oberon/F [OF–94], has been developed from scratch as a commercial environment for component software and compound documents. Oberon/F features abstract classes *View* and *Container*. Any concrete object of type *View* can be embedded in any container. Oberon/F will further integrate both industry standards, OpenDoc and OLE. It is a development tool making all the achievements of Oberon available to the software industry.

The expressions framework to be described later has been implemented in Oberon V4 but the concepts are expected to be portable to System-3 and Oberon/F. Such a port has not been done, since no gain can be seen (except for proving the feasibility). The only feature not available in Oberon V4, which would have been useful, are System-3 *Libraries* or Oberon/F *Stores*, which simplify externalization and internalization of documents with multiple references into a data graph. A port to Oberon/F would also allow for interaction with software based on OpenDoc or OLE.

The restricted compound document model of Oberon V4 turned out to be an important source of inspiration rather than a problem: Text Elements did not only prove to be sufficient, but they triggered the fundamental idea to use Text Elements as tokens of a formal language as described in [Weck–94].

## 2.3 Compound Documents

### 2.3.1 Compound Documents

*Documents* in general are entities that can be visualized, edited, and stored to files. Traditional documents feature one model, for instance text, graphics, or a bitmap. The contents of such a document are homogeneous. *Compound documents* generalize this notion of documents. They may contain arbitrary objects, not only those belonging to one specific model. Such objects within a compound document are called *parts* of that document. The document assumes the role of a *container* for its parts. Parts themselves may be containers, too.

**A Note On Terminology:**

The term *part* has originally been introduced in [OD–94]. It was changed to *component* later. Within this thesis *part* shall still be used for document components to distinguish them clearly from *software components* (see Section 2.2).

The fact that industry tends towards using the term *component* for both software components and document parts is symptomatic of a limited conception of compound documents. Document parts are seen as user interfaces of application program-like software units, but parts can be seen more generally as objects that can be accessed by different software components. This can be elaborated for enhanced usage of compound documents (see Section 2.3.2.).

As an option, containers can support intrinsic data, an extra model managed by the container itself. The parts are inserted in this model. In a text container for instance, parts are treated like special characters. They have a position in the text and can be moved by the usual text editing means. However, parts do not need to recognize in what kind of container they are included and whether extra intrinsic data is featured.

Previous efforts aiming in the same direction led to object-oriented text systems and editors like [ABMW–88, Calder–90, Vetterli–91, Szyperski–92]. These tools allow to include objects of different types into documents. In the beginning, this concept was designed with desktop publishing in mind, i.e. for creating documents for human perception. [Schär–91] describes the inclusion of mathematical formulae into such an editor.

### 2.3.2 Enhanced Usage Of Compound Documents

The compound document concept allows for extension in two dimensions: parts and containers. The bare compound document idea draws a separation line between these dimensions. Parts shall not have any knowledge about the context their current container may provide, and the container shall not have any knowledge about the individual part's functionality.

Although this total separation has its benefits, we find that better use of the compound document idea can be made if this separation becomes less strict. In some cases it is useful to see container and part as a complementary pair. One example is that of *paragraph control characters* (similar to rulers) defined in [Szyperski–92]: these parts influence the typesetting in their surrounding text container. On the other hand, parts can be seen in the context of interpretable text, as discussed in [Weck–94]. Such parts are used as tokens of a formal language. Still, when used separately, all those parts and containers keep their basic functionality like visualization, persistence, in-place editing (of parts), etc., but lose their particular extra functionality.

The most primitive relationship between containers and parts is that of inclusion, i.e. the logical predicate whether a specific part is included in a specific container or not. This relationship does not depend on the container's and part's types, but it is enough to use compound documents as workspaces.

We used this approach in the expressions framework discussed in this thesis to create a simple expression editor (Section 4.3). However, all those parts can also exist within other, unknown containers. They keep part of their basic functionality like visualization, persistence, etc., but lose the part that depends on the particular context.

### 2.3.3 Compound Documents And User Interfaces

Due to their flexibility and extensibility, compound documents have been used to interface software in various ways since long ago. An example of such usage are dialogs, composed with a document editor ("visual designer"). The parts are input fields, buttons, sliders, etc. Their in-place editing mechanisms provide a user interface to the values associated with them. The document editor is used for interface design. Such *compound user interfaces* are widely spread. Examples are [Gutknecht–94, OD–94, Schneider–95].

A very different example for a user interface based on a compound document is the Finder program [InsMac–91, OD–94] of the Macintosh computer. It represents a disk as a structured "compound document" composed of folders, documents, and applications.

With the industry standards OLE [Brockschmidt–94] and OpenDoc [OD–94] and their accompanying COM and SOM object model definitions, compound documents will be used to integrate the user interfaces of software components. Component-oriented programming leads to independent extensible software systems: only the user combines components, developed by different independent suppliers. An example are MathType-Parts included in Microsoft Word, using OLE and COM. The formula editor and the document editor have been developed completely independently by different vendors.

### 2.3.4 Document-Centered Software

Nowadays, the user's activity is centered around application programs and processes: the software is *process-centered*. Data is to a process, and each process operates only on the data assigned to it.

Process-centered application programs are a package deal, bundling data management and computational functionality. Processes structure both the data space and the space of functionality. Encapsulation makes it impossible to apply the computational functionality of one program on the data managed by another program. To combine the computational functionality of two application programs, the data has to be replicated.

Component software, as seen for instance by OpenDoc, is primarily structured by data collections. Such data collections are implemented by documents. A data items belongs to a document. Software to manipulate the data is bound to documents as needed. Consequently, OpenDoc uses one process per document, not one (or several) per application program. Since the dominating structure of such a software system is implemented by documents, such software is called *document-centered*.

In this thesis we go one step further. Data management and computation software are treated as orthogonal. Processes are abolished entirely. The data is stored within a single address space and can be accessed by all software components.

This software architecture is used by the Oberon system. The loss of safety, caused by abolishing separate address spaces, is compensated for by memory protection at object level. The latter is implemented through a strongly typed programming language.

An extensible system is made up by a collection of many small components, sharing their data. For extensibility it is not feasible that each component manages its own, separate data repository. Instead, data must be managed by a central service. Such a central service can use documents as data repositories. Computational components access documents through a programming interface. Data can be passed and shared by reference. This is crucial for efficiency, since in a system built from smaller components data will be frequently used by different components.

Section 4.1 shows that compound documents are a light weight implementation of data repositories. In fact, most of the functionality needed can be reused from the document framework. Compound documents are already a single answer to three separate problems:

- typesetting: documents can be edited with the purpose of being printed for human reading.
- persistence: documents can be stored into a file.
- user interface: operations can be invoked on document parts to change them.

With the data repository as a central component, workspaces and computing actions are decoupled. The user gains new freedom. Commands can operate on objects in any document and documents can bundle objects of any type. Several independent command components can be used with the same workspace. There is no need for data replication. On the other hand, several workspaces can be used concurrently. With this, data can be structured according to one's own needs. Data can be transferred between workspaces by moving icons. This does not lead to data replication, since only a reference is copied.

Not only integration of various mathematical software components is possible, but also completely different software can be integrated. All the existing document processing tools can be applied to workspaces. A workspace can be sent via e-mail, for instance. Also, workspaces are integrated into documents. The benefits are, for instance, that comments can be put close to objects, graphical output (plots) can be directly related to its mathematical data, etc. Since the data objects can still be used for computation one can view such a document as being interactive. Combination with hypertext facilities opens the road towards interactive textbooks.

### 2.3.5 The Editor Is The User Interface

The user interface metaphor for document-centered software is that of a very powerful editor: the user opens, manipulates, and archives documents. Data used for computation is represented by document parts. Operating on the data means changing the workspace, i.e. to edit the document. Editing operations are carried out in cooperation by the user and the machine: the latter changes a piece of a document according to a command given by the former.

From the user's point of view, there is no significant difference between, say, changing the font attribute of a text stretch or replacing a polynomial by its factored equivalent. In both cases a selection will be made in a document and a command will be applied to it.

Editing operations can be implemented by separate components and be added to the system at any time. There is no limit in principle to their power. The entire system can be seen as an extensible editor.

With most existing mathematical computing systems, the user has to wait for completion of a command, before issuing another one. The principal interaction with an application program and the use of a compound document as the user interface are compared in Table 1.

| process–centered software | document–centered software |
|---|---|
| start an application program | (optionally) open a document |
| type input data or copy from elsewhere | create document part or select existing one |
| issue command | activate command |
| output is written to a log text | output is pasted as a new document part |
| assign the previous result to a variable | – |
| use variable | copy expression from document part |
| copy & edit the input line | copy & edit a document part |
| terminate application program | (optionally) close the document |

*Table 1 : usage of process-centered software vs. document-centered software*

It is desirable to leave it to the user whether the result should replace the input or rather should be inserted without affecting the input. The latter style is quite often used to produce documents that besides presenting results also document how the results were computed. More sophisticated constructions could also be thought of. For instance, an entire equation could be produced as output, relating the input and the result. If an integration command is applied to $tan(x)$, the result could be presented as $\int tan(x)\,dx = -ln(cos(x))$.

However, the author's experience with various experimental implementations showed that it is most convenient if just the selected document area is changed. Commands doing so are editing operations, like those known from document processing. Removing the original values is not a severe restriction: the user can copy the parameters and operate at the place where the result is wanted.

# 3 Design Of A Framework For Mathematical Component Software

Generally, mathematical software can be extended in three dimensions: expression types, algorithms, and user interaction. Only a few relations exist between the dimensions of extension. Algorithms, for instance, need to have some knowledge about the objects they operate on, and the editor needs to be able to typeset expressions of any type. The fundamental abstraction to be used by all extensions is that of a mathematical expression. Therefore, in the center of the framework is the definition of abstract expressions (see Figure 1). Algorithms are implemented as library procedures. These can be accessed from other components and used to build higher order algorithms from existing ones.

## 3.1 Expressions Are Directed, Acyclic Graphs Of Objects

It must be possible to introduce new expression types, as mathematics is a very large and rapidly expanding discipline and the set of notations and associated properties will never be complete. Therefore, typical mathematical software relies on a very open representation of expressions: named functions with zero or more parameters. The mathematical semantics are coded into the function name. The function's parameters are the subexpressions the object's value depends on. To typeset a function a pretty printer is used. Of course, the latter must know the function name to perform any displaying other than a default.

Presenting expressions through named functions or similar mechanisms has two major disadvantages. First, name clashes cannot be excluded. Two independently created extensions may use the same name for two different objects. Secondly, as illustrated by the pretty printer example above, the

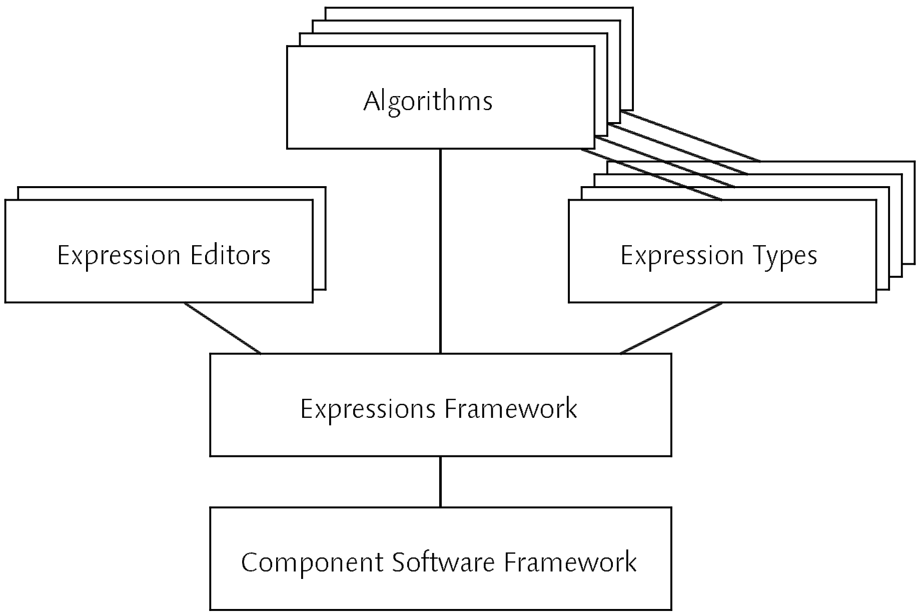


*Figure 1: the structure of mathematical component software*

names need to be registered with other components to be recognized. This results in case analysis instead of generic, object-oriented programs.

Types can be used instead of names: an object has a type (belongs to a class) which can be distinguished from any other type. This property is guaranteed by the system. Name clashes are avoided. Registration with other components is not necessary since behavior can be bound to objects. Extensibility is granted by subtyping.

Generally, an expression consists not of a single object but of a rooted graph of objects. The graph's structure is the same as with functional representation: the root node defines the mathematical properties of the outermost structure of the expression and refers to subexpressions. The graph is directed and acyclic. For each node the outgoing vertices are ordered, since, in general, operators are not commutative.

Each object representing a node is of a type extending (subtyping) a general, abstract expression type. This allows for extensibility with new expression types. Expression graphs are generic, inhomogeneous data structures. The basic type of all expressions is defined in the expressions framework to be extended by concrete implementations.

## 3.2 Expressions Are Immutable

Expression objects, once created, need to be immutable. This drastic restriction is a requirement of independent extensibility.

Expression objects are used as values, not as variables. References to expression objects will be shared by different components. Independent extensibility implies that it is impossible to control which components share references to an expression object. Consequently, if one component changes the value of an object, it may affect other components in an uncontrolled way. In this situation, a component could not rely on expression objects to have constant values. It needed to generate a private copy, and a huge overhead would be the result.

To avoid this, a component which changes an expression has to create a new instance. This could be done by convention or it could be enforced by the framework. The latter is preferable. The framework has to offer an abstract data type for immutable expression objects.

All aspects being considered, the immutability restriction has more advantages than disadvantages. With asserted immutability, data replication can be reduced to a minimum. If an expression graph is to be copied, only nodes actually changed need to be replicated. All unchanged subexpressions can be included by reference.

Representing expressions by immutable DAGs has proven useful already, for instance in Maple [CGGLMW–91].

Also, abstract data types exist that enforce immutability of objects. An example are immutable texts implemented as *Ropes* [BAM–94]. Whenever a text represented by a *Rope* is changed, a new *Rope* is generated to represent the new text. This shows the disadvantage of *Ropes*: when a sequence of changes is applied, many objects are generated unnecessarily. In Section 5.1 a solution is suggested which allows more efficient manipulation.

## 3.3 The Typical Operation On Expression Graphs

Most operations executed on an expression graph follow a typical pattern. First, a node is identified which represents the subexpression to be changed. During this identification process the path leading to the node is recorded. Next, a new graph is built representing the expression resulting from the operation. Thereby, every node, on the path recorded before, is cloned and the new subexpression is included. For further illustration see the complete example of a generic substitution operation in Section 5.4.

The above procedure needs to work based on the abstract expression type only. The abstract expression type is the common interface of orthogonal extensions: types and algorithms. It allows algorithms to operate on subexpressions of expressions of unknown type.

Some operations will not change just one node but several within a graph. Consider for instance a command that can compute with rational numbers. Such a command will traverse the entire graph and simplify rational subexpressions as far as possible. Such subexpressions may be found in multiple places.

$\frac{7}{10}+\frac{2}{15}+\int_0^x \frac{(3+7)\cdot\sqrt{t}}{2}\,dt$, for instance, can be simplified to $\frac{5}{6}+\int_0^x 5\cdot\sqrt{t}\,dt$.

A consideration on efficiency may be in order: dealing with immutable expression graphs requires some overhead. Computing algorithms, however, have to use expression graphs only to interface the framework. Internally, they may build their own data structures suited for the particular algorithm. This must be possible anyhow, since the efficiency of an algorithm may depend heavily on the data structures used. For example for polynomials, [Stoutmyer–84] lists a wide variety of representations, each of them suitable for some algorithm.

## 3.4 Equality, Canonical Forms, And Redundant Representations

### 3.4.1 Equality

It must be possible to test for equality of two expressions without knowing their actual type. As an example, consider a substitution algorithm which replaces all instances of a particular expression in a graph. Such an algorithm can be implemented generically if a generic equality test is available (see Section 5.4 for a complete implementation).

Equality must be defined carefully. Mathematical equality may even be undecidable. Examples of such cases are given in [Richardson–68] and [Richardson–69]. Only structural equality is always decidable, i.e. two expressions are known to be equal only if they are represented equally. In general, it is not possible to test for two arbitrary expressions, whether they can be transformed to have an identical representation.

Structural equality can be defined recursively as follows: two expressions are equal if the two root nodes are equal and if the subexpressions are pairwise equal. Whereas the latter can be determined generically by recursive application of the test, the former requires to call a procedure bound to one of the nodes.

The entire test can be performed in time linear in the number of nodes. For this, it must be assumed that the lists of subexpressions of the two candidates are in the same order. Commutativity cannot be tested; $a+b$ is not equal to $b+a$. Even if commutative subexpressions would be marked, testing all combinations would lead to a non-linear algorithm. The solution is to generate a canonical normal form prior to comparison. Canonical normal forms are unique representations. Two expressions which are equal have identical canonical normal forms. Each expression object must offer a method creating a canonical normalized representation of itself. This method must have a way to order commutative subexpressions uniquely. For this, a global ordering on expressions must be defined.

### 3.4.2 Ordering Expressions

To define an ordering, an integer value is associated with each object. Two objects must get the same value if and only if they are equal. This property is not met by a hashing function: it could assign the same value to two non-equal objects. A central stamping service based on an incremented counter does not work either: a new value would be assigned to every object, regardless of any equality to previous objects.

Consequently, all objects which have a value assigned already must be registered. If for a given object an equal one has been registered before, the same value is assigned. Otherwise, the object is registered and receives a new value. Searching the registry can be sped up by hashing. The integer value needed is derived from the index into the registry.

To derive a canonical normal form, the above procedure has to be executed recursively bottom up: first all subexpressions are registered and converted to canonical normal form, than the expression itself is treated.

### 3.4.3 Eliminating Redundant Representations

A further application of equality testing is to eliminate redundant expression objects. Since expression objects are asserted to be immutable, there is no need to replicate equal data. Eliminating redundant expression objects can reduce the memory requirements dramatically. Not only entire, large subexpressions need to be stored only once, the same holds also for small but frequently occurring objects, like symbols.

Canonical normal forms and elimination of redundant expression objects are independent concepts. Any equality relation can be used for the latter, for instance equality of representations. This separation becomes important when the data is not only used for computation but also for presentation within documents. It must be possible to include equivalent, yet differently presented expressions into a single document. If a document contains $a+b$ as well as $b+a$ neither of the two must be changed.

Redundant expression objects can be eliminated with a similar procedure as to compute canonical normal forms: expression nodes are registered to define a unique object, representing each equality class. Searching for possibly equal objects in the registry is sped up by the same hashing mechanism as used for canonical normal form computation.

It should be noted that the procedure outlined in Section 3.4.2, computing canonical normal forms, also eliminates redundant representations.

### 3.4.4 Global And Local Scopes

The table used to compute canonical normal forms and/or to eliminate redundant objects interferes with garbage collection. (Garbage collection has been identified as a must in any extensible system in Section 2.2.4.) The marking phase of garbage collection must not follow the references in the table. Otherwise none of the registered objects would ever be considered as unused. Furthermore, references to objects being collected must be removed from the registration table.

One solution is to integrate the registration table into the garbage collector. Another solution is not to use a global table, but several smaller ones with limited scope. Usually, a canonical normal form is required in a specific context and for a specific set of objects. A table can be generated explicitly for this scope.

Enforcing global elimination of redundant objects has some advantages though. Maple [CGGLMW–91], for instance, asserts that references are shared whenever possible. This saves memory, since every value is stored only once. Furthermore, it allows efficient equality tests. Comparing the references suffices. The disadvantage is that every expression generated or entered by the user must be immediately transformed into its canonical normal form. Also, an expression read in and echoed to the user may look different from what the user had entered.

If the same data structure is to be used for computation and representation within documents, this must not happen. Redundant expressions may only be eliminated if their representation is equal. This in turn makes it impossible to base mathematical equality testing on pure reference comparison. Hence, for our purpose a global registry does not pay of.

## 3.5 Generic Drawing And Editing

The widely used Model-View-Controller Separation (MVC) [KrPo–88] allows to extend a system with view and editing components. The components implementing model, view, and controller of an editor form a hierarchy. The model component is is not concerned with drawing or editing aspects. Components implementing drawing or editing commands can be added later to the system. In particular, different view components can be used to achieve different presentations of the same model.

To display the state of a model, the view component interprets the model's data. If the model is an inhomogeneous data structure, the view component has to perform case analysis. If the model is an extensible data structure, this can no longer be done.

Expression graphs are an extensible data structure. The MVC model can be used in a restricted form only: the drawing procedures must be bound to the individual expression types. However, there is still room for some extensibility. On one hand, expressions can be drawn within different environments, for instance in a separate window or in a compound document part. The definition of these environments can be left to extension components, similar to the view components in the MVC model. On the other hand, abstract draw ports can be used to bind the drawing device lately.

In [Kajler–92] a user interface for symbolic computation is presented that requires to extend the model and view components in parallel. Case analysis is possible, but to extend the system, several components must be changed.

### 3.5.1 Bounding Boxes

Expression graphs are defined recursively. Each node may refer to several subexpressions. When a node needs to generate a two-dimensional graphical representation, this representation needs to include the representations of its subexpressions. The spatial extension of the latter depends on the type and value of the subexpression. Therefore, a generic mechanism is required to arrange these objects.

A minimal abstraction for an expression in two-dimensional space is its bounding box. It is computed recursively. The bounding box of an expression depends on the bounding boxes of its subexpressions, and so on recursively.

Whereas expression graphs are DAGs the box graph must form a tree. The appearance of an expression may depend on the nesting depth within the entire formula. For instance, in deeper nesting levels smaller fonts may be used. With large expressions their graphical representation may leave out details to give a better overview about the main structure. Consequently, subexpressions represented by the same node may have different appearances (see Example 1).

**Example 1**

$$x + \frac{1}{x + \frac{1}{x + \frac{\cdots}{\cdots}}}$$

The time to build a data structure containing all bounding boxes is linear in the number of boxes and also in the output size. In the worst case the number of boxes is exponential in the number of nodes in the DAG. Still, the output will usually be kept within reasonable size, for instance with the help of elisions.

Building the box data structure instead of drawing the expressions on the fly is a matter of efficiency: without using additional storage, the algorithm for drawing an expression would be non-linear. In the first step the bounding boxes of all subexpressions are computed recursively. Depending on the result of this computation the subexpressions can be placed. In the final recursive step all subexpressions are drawn. This involves recursive computation of all the bounding boxes of their subexpressions once more, leading to non-linear behavior.

### 3.5.2 Binding Power

One more issue must be abstracted: the binding power of operators. When drawing an expression for each subexpression has to be decided whether parentheses must be put around it or not. To allow for a generic decision, each expression carries a value specifying its binding power on a global scale. If this value is lower than or equal to that of the surrounding expression, parentheses must be drawn.

As a consequence, some extra parentheses will be drawn, depending on what had originally been entered by the user. Note that this behavior is desired for expressions within documents.

For instance, the expression $a+(b+c)$ is represented internally by two objects (two sums). Following the above rule, the parentheses are drawn. Conversely, the expression $a+b+c$ is represented by a single object and thus no parentheses are drawn.

The decision whether to put parentheses around a subexpression is programmed explicitly within the box generation procedure of every expression type. This allows for different binding power on the left hand side and the right hand side of an operator. As an example consider exponentiation, which has higher binding power on the left hand side than on the right hand side. Therefore, parentheses are required around the basis but not around the exponent: $(a \cdot b)^{a \cdot b}$.

Independent extensibility requires to provide a fixed set of binding power values. Consider two independently created expression types being combined in one expression. To determine whether parentheses must be used, the two binding power values are compared. The result of this comparison must be anticipated by the creators of the two components, though they are nor aware of each other. Consequently, the component creators need a globally fixed set of possible values to guide them.

A small scale of values appeared to be sufficient for the prototype. Five values are defined (strongest to weakest binding): *atomic*, *exponential*, *product*, *sum*, *none*. Parentheses are never needed around an atomic value, but always around a value with no binding power. The three other values correspond to the known arithmetic operations.

### 3.5.3 Editing

The box tree data structure can be used to create a generic, graphical expression editor. Such an editor will display the expression to be edited and also store the box tree generated during this process. This data is sufficient to implement an expression selection mechanism. For any coordinates pointed at with the mouse the innermost bounding box can easily be identified. This box refers to the expression it presents.

After selecting a subexpression it can be replaced by another expression using the typical operation on expression graphs, specified above. Starting from a collection of predefined expressions (a palette) arbitrary expressions can be composed using this copying mechanism. The resulting editing model is similar to that of templates as defined by [Soiffer–91].

The editor sketched does not depend on concrete expression types. It is completely generic, allowing for independent extensions with expression types.

The interface between expression types and editors is very small. It consists of the possibility of creating bounding boxes and drawing on an abstract port. Expressions to be displayed are known to be structurally complete. This is guaranteed by the abstract data type for expressions. Incomplete expressions cannot be represented by that data type.

Model-View-Controller Separation [KrPo–88] allows to display and edit the same model in separate views at the same time. Usually, this requires synchronization of the different views.

As discussed before, expression objects need to be immutable. Even an editor will not change the value of objects, but create new ones. Therefore, synchronization of different views is not necessary. Other views, than the one edited, keep the reference to the original object. Only if expressions are embedded into another model, e.g. in a compound document, and if this higher order model is visible in multiple views, synchronization is required at that higher level. A wrapper of the expression editor, the document part in the example, is responsible for this. The expression model itself does not need to support synchronization.

### 3.5.4 Textual Editing

The editor described above is purely graphical. It must be operated with a pointing device. It is not possible to enter expressions by typing on the keyboard. The latter is demanded by experienced users as an alternative.

Presenting expressions as text in an independently extensible system presents some problems. Names and special characters have to be used to denote operations, values, and functions. This inevitably reintroduces the problems of name clashes, successfully avoided so far. Two new expression types, introduced independently, may happen to use the same name in their textual representation. However, the problem occurs only at the top level, in the user interface. It does not affect the security of the system. It must be left to the user to resolve such name clashes: if the input is not interpreted in the expected way, it can be corrected by graphical, object-based editing.

Two technical problems have to be resolved, to allow for textual editing. Text entered must be parsed and transformed into expression graphs. Conversely, expression graphs must be able to provide a textual presentation of themselves. These mechanisms must be implemented generically, i.e. for arbitrary expression types.

It is relatively difficult to make the parser extensible. One solution is to base it on interpretation of an abstract grammar, like for instance done for the Orm System [KLMM–94]. Extending such a grammar to deal with new extensions requires changing multiple components. This should be avoided. Instead, new objects should install a procedure to be called by the parser. Since this procedure is to be implemented by every extender, it should have a simple interface and should have to perform a simple operation only.

For a simpler approach a basic set of expression types is defined. The language is closed and allows to represent expressions built from this basic set. Genericity can be achieved by using named functions to represent arbitrary expression objects as text. (Such named functions are one type of the basic set of expression types.) To internalize a textual expression it is first parsed into an expression graph. In a second step objects can be replaced by more precisely typed objects. The transformation procedures installed are simply transformers on expression graphs. They do not have to deal with text directly, neither with syntax errors, since completion of the first phase asserts that an expression could be parsed.

Not all function objects need to be replaced. The sin function, for instance, can be taken as is, whereas an *Integration* function will be substituted by the properly typed object.

Conversion of an expression graph to text works in a similar way: with expression types a method has to be implemented which generates an equivalent representation using types from the basic set only. In a second step, this new representation can be transformed to text using case analysis.

Section 4.3.1 discusses the details of a generic text-based expression editor. The language implemented is based on the following expression types: arbitrary precision integer numbers, symbols, named functions with an arbitrary number of arguments, some binary operators (+, −, *, /, ↑), indices, and sets. The first three types would be sufficient; the latter three have been included for convenience during text editing and because of their frequent occurrence.

## 3.6 The Editor As User Interface To Computational Commands

The generic editor sketched above provides the base for a simple user interface to computational commands. The computational data is represented within documents. Operating on the data means to change a document. Computational commands are presented as editing commands. From the user's point of view, there is no difference between, say, changing the font attribute of a text stretch or replacing a polynomial by its factored equivalent. In both cases a selection will be made in a document and a command will be applied to it. User interfaces for mathematical software based on a similar idea are discussed in [ABMW–88, Sydow–92].

Editors and computational commands are orthogonal extensions. The framework must define an interface, allowing the editor and the computing commands to exchange information. This information exchange is either retrieving the currently selected expression, or replacing an expression with a computed result.

It must be possible to keep information, only available when retrieving a selection, for usage during expression replacement. For instance, if more than one editor is active, the editor which had delivered the selection needs to be determined. Or, if the computation is done concurrently (maybe on a remote server), the selection state of the editor might have changed in the meantime. Therefore, when retrieving a selection, an object of a special, extensible class is returned. A replace method is bound to this object.

# 4 Document-Centered Mathematical Software

## 4.1 Compound Documents Are Better Workspaces

Compound documents can implement workspaces as defined in Section 2.1.2. Document parts are associated with the objects to be included into the workspace. Parts whose main purpose is to be a reference to an internal object will be called *icons*. Such icons within compound documents do not only implement all the requirements for workspaces, they can do this better than name bindings, mainly because of the absence of hidden state, and the potential of graphical presentation.

### 4.1.1 Icons

A compound document part always refers to an object. From another point of view one can say that the object is represented by the document part. It is because of the document part that the object becomes visible and accessible for manipulation by the user. Document parts with this use in the foreground are called *icons*.

**Definition:** An *icon* is a visible entity (here a compound document part) that refers to an internal object or a data structure. It can be used by the user as a *reference* to an otherwise not accessible object. The way the icon presents itself graphically is called its *appearance*.

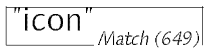
#### Reference And Appearance Are Well Separated

The above definition focuses on an icon's referencing character, which is separated from an icon's appearance. Due to this separation the latter is very flexible: depending on the purpose and the kind of the object to be abstracted, the icon's appearance may be some predetermined picture (like with traditional icons), a graphic indicating the object's type, or a graphic indicating the actual state of the object. In the case of the latter, the state may be graphically editable.

The important advantage of icons over representing objects by names is that the appearance of an icon is not dictated by the reference mechanism. The appearance of a name is completely determined as a certain sequence of characters. If a single character is changed, a name will fail to work as a reference. The appearance of an icon is a property that can be chosen freely without affecting the reference to the object. The separation of reference and appearance is discussed in detail in [Weck–94].

## Example 2

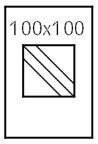
An icon abstracting the answer to a query in a text data base:

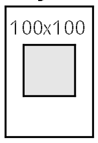


In this example the appearance reflects explicitly the type of the object abstracted (i.e. *Match*) and part of it's state (query was *icon*, answer size is *649*). The object represented by the icon is a set of text positions in the database.

## Example 3

An icon abstracting a numerical matrix:



In this example the appearance reflects the object's type (matrix) only implicitly but gives explicit information about the matrix's dimensions (100 by 100) and its tridiagonal character. The object referred to is a two-dimensional array of complex numbers (as opposed to a full matrix represented as .)

**Example 4**

An icon abstracting a data structure representing a mathematical expression:

$$\int_0^{\pi} \frac{\sin(x)}{\cos(x)}\, dx$$

In this example the appearance is reflecting the complete state of the abstracted object, i.e. the exact expression represented by that object.

Depending on the space needed for the complete presentation of the object, the icon may display a reduced or collapsed version (like in the matrix example above). This is an important issue: on one hand the icon should present as much information as possible. On the other hand the user wants to be able to see as many icons within a document. Therefore individual icons must not consume too much space. Choosing the right amount of data to be represented by an icon is a major design decision. It may even be reasonable to allow the user to influence the granularity of representation. In any case, if the icon does not reflect the entire state of an object, additional commands should be provided to zoom into the data or to display it in a separate window.

**Example 5**

In sums with many terms or in matrices elision is used to reduce the number of objects displayed:

$$x^{10} + x^{9} + x^{8} + x^{7} + \underset{\cdots}{6} + 1 \qquad \begin{bmatrix} s & 1 & \dots & 0 \\ 1 & s & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & s \end{bmatrix}$$

The following expression exhibits elision of subexpressions because of their depth:

$$\frac{1}{1 + \frac{1}{1 + \frac{\dots}{\dots}}}$$

It should be noted, that in our implementation elision only affects the graphical presentation.

An icon is not an expression. It is a document part referring to an expression. Because of that it can carry more information than the expression it refers to. In particular, additional information about the graphical presentation within the icon can be stored with it. The granularity of the presentation is an example of such information.

Since icons carry direct references to objects they can retain more information than pure graphical presentations would. This is important when results of earlier computations are used as input to further computing. The full mathematical information of the computed result is retained. This may include type information and others which would be lost if the intermediate representation would have to be purely graphical. Furthermore, the graphical representation of an icon can use elision or other mechanisms to reduce the amount of space needed without actually loosing the information. These advantages are a direct consequence of the separation of reference and appearance.

### 4.1.2 Compound Documents With Icons Form Workspaces

Icons as compound document parts can be selected as input for commands, and they can be generated as output by commands. Such output may either be pasted to a global insertion focus or replace the input data. Thus, compound documents can serve as user interfaces for computational commands. From the compound document point of view such commands are equivalent to any editing operation (see Section 2.3.5).

Compound documents together with icons can be used as workspaces. To prove this statement, it has to be shown that all the requirements for a workspace stated in Section 2.1.2 are met:

- The workspace is the compound document itself. The objects in the workspace are those being represented through icons within the document.
- *Insertion, replacement, or removal* of an object is done by pasting, replacing, or deleting the corresponding icon, using the normal editing operations of the document editor.
- To get the *overview of all objects within the workspace* scrolling through the document is sufficient, since all references are visible as icons. No hidden state is involved.

- To *refer to an object* in the workspace the corresponding icon is selected using the selection mechanism of the compound document editor. Operations use this selection to retrieve their input data. If an operation depends on more than one object, multiple selections can be used or the objects can be composed to form a contiguous document section (see Section 4.1.4).
- Icons *display the objects* they represent directly and in-place. No action of the user is required to make an object visible.
- The objects in the workspace are *externalized and internalized* by the normal store and load operations for the compound document, since they are part of it.

### 4.1.3 Advantages

Implementing workspaces with compound documents and icons has several advantages over using name bindings. These result mainly from icons being visible references not involving hidden state. Workspaces can be *manipulated directly*. This is an improvement over manipulation through programming languages, as already stated in [Shneiderman–83].

There are three main advantages of icons in documents:

- direct representation and manipulation of the workspace's state
- more direct information about the individual objects
- the possibility of changing objects directly

#### True Document-Centered Representation

Implementing workspaces with documents independently of computing programs leads to the freedom and openness of document-centered software (see Section 2.3).

#### Higher Transparency Of Workspaces

Icons refer directly to the objects they stand for. There is no internal table of name bindings to be kept consistent with the user's knowledge. The user does not need to memorize previous assignments and cannot accidentally overwrite a name binding and lose information in this way. This distinguishes our approach from many others (e.g. Maple's *Worksheets* [CGGLMW–91] or Mathematica's *Notebooks* [Soiffer–95]) which use documents only as editors, allowing to manipulate workspaces, implemented elsewhere (see also Section 6.2.).

Mnemonically chosen names can also present information to the user. Such information has to be presented in a different way, when using icons. For instance, names can express further attributes of an object or relations between several objects. As an example consider a function named *f* and its derivative, which can be called *df*, or *fprime*. Another example is a set of expressions, being called *exp1*, *exp2*, *exp3*,... to indicate their relation. Workspaces implemented as documents offer other possibilities to present such information. The placement of icons within the document can be used to express relations between objects. Related icons can be collected in a separate paragraph or within a graphical box. Arbitrary extra information can be presented as text or as picture close to an icon.

Objects are stored and loaded together with the document holding references to them. This gives a very light-weight implementation and a simple user model. Storing only part of the workspace's contents can be done by copying the objects in question to a separate document and storing that.

Icons present a user interface to a system's garbage collection, where such a mechanism exists. Since objects still accessible consume a visible resource (space within the document), the user will naturally clean up on a regular basis and remove icons no longer needed, allowing the corresponding object's storage and possibly other bound resources to be released. The document metaphor gives a clear feedback about what is part of the workspace. Membership within workspaces can be manipulated directly.

### More Information About Objects

The visual appearance of an icon is not restricted to primitive pictures but can give more or less detailed information about the object in question. A mathematical formula e.g. can be typeset within an icon. Such a graphical appearance can present much more information to the user than a simple name can.

As discussed in Section 4.1.1, the graphical presentation needs not to show the exact data. For instance, elision as defined by Soiffer [Soiffer–91] may be used to reduce the space needed for drawing.

Icons are references to structured expression objects instead of pure graphics. This results in a constrained editor: only well defined expressions can be created. Drawing as well as computation can rely on this, simplifying their task. Furthermore, expression objects can carry more information than pure graphics can. This is of particular importance whenever a result of a previous computation is to be used and for instance type information needs to be retained and passed.

### Icons Can Be Reactive

As a further option, the visual objects can be made reactive and consume user input. A mathematical expression, for instance, can be represented by a document part implementing a complete expression editor. Such a part assumes two roles at the same time: that of an icon in the sense of our definition and that of an editor. Some expression editors are already available as document parts (see Section 6.4).

Mathematical expressions are defined with a recursive structure. In the graphical presentation of an expression a subexpression can be selected with a pointing device. Such a selection can be used as input for computing commands on the same basis as an entire icon.

### Sessions Are Not Needed

Together with the management of hidden state the concepts of history, sessions, and therefore also session control can be given up. Sessions are needed as scopes for name bindings and to steer a program's start and termination. In our model name bindings do not exist. Loading and unload ing of programs is completely decoupled and organized by the environment. Typically, the code necessary to execute a specific command will be loaded into the system when its execution is requested for the first time, and it remains there until unloaded either by the system to free resources or by the user, e.g. to force loading of a new version. The only aspect of session control still needed is a possibility to interrupt an ongoing computation.

From the user's perspective, the protocol of a session may be of interest. It describes a sequence of steps to reproduce the results. Such a protocol can be presented within a document, too. (Some of the examples given within this thesis can be seen as such protocols.) Since documents can be edited arbitrarily, a special mechanism to enforce the correctness of such a protocol is needed. This can either be implemented by an explicit command to recalculate the individual steps, or by a construction similar to spread sheets. Both such mechanisms are discussed as scripting mechanisms in Section 4.2.3 and 4.2.4.

## 4.1.4 Composing Parameters – Using Icons In Context

To execute a computational operation as an editor command requires the parameters to be selected beforehand. Commands operating on several objects require multiple selections. These selections need to be ordered, since many commands will not be commutative on their input objects. To give the user feedback on the ordering, selections need to be marked differently.

An alternative can be developed if icons are seen within their context, as has been suggested in Section 2.3.2. Without loss of generality and for the sake of simplicity we will restrict ourselves to icons in textual context. These can be interpreted as tokens of a language. Icons can denote subexpressions within a text that represents an expression. A text relating all the parameters of a command can be composed using drag & drop. The operation's results can replace the input parameter text, which is only a copy. This fits well into the editing metaphor discussed before (see Examples 6 and 7).

**Example 6**

Assume that the integral $\int_c^b f(x)\,dx$ is to be subtracted from $\int_a^b f(x)\,dx$. Instead of selecting the two operands in a specific order (subtraction does not commute!) before activating a subtraction command, the following expression is generated at the place, where the result is wanted:

$$\int_a^b f(x)\,dx - \int_c^b f(x)\,dx$$

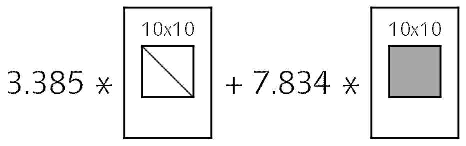
The minus between the two expressions can be typed on the keyboard, whereas the two integrals are copied using drag & drop. Selecting this expression and executing an evaluation command (Maple.Combine) replaces the text stretch with the expected result:

$$\int_a^c f(x)\,dx.$$

**Example 7**

Not only operators but also other objects, like numbers, can be included into the input in textual form.

The following is an example for a weighted sum of two matrices:



### Generalization

Formulae can be represented in various ways. It can be a single icon, plain text, or any mixture of these. It is up to the user which representation to use. The user can compose formulae in the most convenient way. The parser component has a key role. It separates the concern of representation from the rest of the software. All the possible representations are analyzed by the parser. Regardless in which form the input is, a single expression object will be returned.

## 4.1.5 Operating On Subexpressions

An icon allows selection of a subexpression within the expression it represents. Such a selection can be used as input to computational commands, see Example 8.

Our experience has shown that it is the most natural way to let every command replace the input with its output. This works well for both operating on a text stretch interpreted as an expression sequence and operating on a subexpression selected within an icon. (Alternatives to presenting commands as editing operations are discussed in section 2.3.5.) The interface abstracting the different selections and the corresponding replacement actions is a central service to be used by all commands. The scripting facilities discussed in Sections 4.2.3 and 4.2.4 create a virtual selection. Therefore, all commands can be used in scripts, too.

### Example 8

Given the following expression:

$$\left(\mathrm{f}\left(x^3-4\cdot x^2-3\cdot x+18\right)+a\right)\cdot\left(\mathrm{g}\left((x+4)\cdot(x-3)\cdot(x+5)^3\right)+b\right)$$

Consider we want to factor the parameter of function *f* and to expand the parameter of function *g*, leaving the rest of the expression unchanged. The subexpressions to be expanded can be selected individually to apply, for instance, Maple's factorization and expansion commands, leading to the following result:

$$\left(\mathrm{f}\left((x+2)\cdot(x-3)^2\right)+a\right)\cdot\left(\mathrm{g}\left(x^5+16\cdot x^4+78\cdot x^3+20\cdot x^2-775\cdot x-1500\right)+b\right)$$

If we had applied the expansion command to the entire expression, the result would not be as desired. The outermost product would be expanded, but the function parameters would be left untouched:

$$f\left(x^3-4\cdot x^2-3\cdot x+18\right)\cdot g\left((x+4)\cdot(x-3)\cdot(x+5)^3\right)+$$

$$f\left(x^3-4\cdot x^2-3\cdot x+18\right)\cdot b+a\cdot g\left((x+4)\cdot(x-3)\cdot(x+5)^3\right)+a\cdot b$$

### 4.1.6 Concurrency, The Client Server Model, Icons To Represent Upcoming Results

The client/server architecture is a general abstraction to present concurrency of computation to the user. It can be used to model computation on a remote server or by a parallel process, and allows for integration of legacy software.

The document-centered user interface presents software as editing operations. The user interface turns into a multi-document editor. Editors in general are single threaded programs. This is to avoid inconsistency of the data caused by interference of parallel threads, which otherwise would lead to non-negligible investments into synchronization and mutual exclusion. This works well only, as long as the user executes single editing operations and waits for their completion before issuing the next command.

As long as only simple editing operations are considered, it can be assumed that the operations terminate quickly. A sequential programming paradigm, blocking the interface during computation, is suitable in these cases. Symbolical or numerical computations can take up to several minutes or even hours. Generally, their duration cannot be estimated in advance. Therefore, the user interface must not be blocked.

When a result from a server becomes available, it needs to be represented by a document part. Three possibilities exist to make a new part available to the user:

- the part can be pasted into a new document which is opened at the same time.
- the part can be pasted into the document the user currently deals with (e.g. at the current input focus).
- the part can be appended to a specific document, e.g. a log text.

Following good user interface conventions the first two possibilities are ruled out: it should be considered bad design if a document changes (or a new window appears) unexpectedly, i.e. not as an immediate consequence of a user action. Using a log has the advantage of automatic documentation of the actions taken, but it requires to explicitly copy every result to its final place.

It is better to change the document upon launch of a command to represent the upcoming result as a part of it. This gives the user feedback with less surprises: the object that will represent the result appears immediately when the command is issued. A visual cue can identify it as the representation of an upcoming result.

In [LiSh–88] mechanisms of this kind received the name *promise*, which we want to adopt here.

Icons can be pasted into a document immediately when the command is issued. Already before they get their final value assigned they represent a handle to the upcoming result. This can be used to compose new expressions and to schedule further commands for later execution.

In contrast to promises in [LiSh–88], which are just names, the icon's appearance can inform the user about the computation state. In [Weck–94] a user interface using this idea is described: an interface to a text searching engine, giving access to the Oxford English Dictionary [Fawcett–89, OED].

Icons used as promises can even be used to control the server's operation: if the user removes the last instance of a promise (by deleting the icon from the document), the corresponding (future) result becomes inaccessible. Hence, the server might as well stop its computation immediately.

In a garbage-collected environment, proper implementation of such a mechanism requires that the object, which represents the upcoming result, performs an action before it is removed. For this, the garbage collector needs to call a method, which can stop the server. This mechanism, that allows a special method to become active before removal of an object, is commonly known as *finalization*.

## 4.2 Scripting

With the user interface model discussed so far only step by step manipulation of data is possible. Scientific computing usually requires an easy to use mechanism, allowing the composition of several commands to be executed as a *script*. Scripting is a form of programming. Its distinguishing features are:

- ◦ scripting focuses on the combination of existing operations rather than on efficient implementation of algorithms.
- ◦ script creation must be quick and easy.

Due to the former property fast execution of scripts is not an important issue. Scripts can be executed by a language interpreter and need not be compiled. This in turn simplifies script creation. Most scripts are ad hoc programs performing a particular task on a collection of input data.

Apart from improving execution speed, the purpose of compilation is to detect programming errors as early as possible. In the case of scripting this early detection may be less important. On one hand, scripts are supposed to be very simple and easy to check manually. For instance, most scripts will not use control structures at all. On the other hand, scripts usually have a short life time.

Scripting can be implemented in four different ways within a user interface based on document editing. Two of them are based on preexisting external mechanisms: the system's extensibility with new commands, and existing scripting components of the editor or its environment. Two further approaches use the compound document concept. All four approaches are introduced below. Section 4.2.6 discusses their advantages and disadvantages.

### 4.2.1 Approach 1: Command Programming

The natural way of programming within a component software environment is implementing new components. An editor-based user interface is most naturally programmed by extending it with new editing commands. A script can be implemented as such a new command.

As the programming mechanism of a component software environment has to be powerful enough for general programming tasks, the remaining question is whether it is easy enough to use. Unfortunately, this is a very subjective, almost a religious issue. The Oberon System, for instance, has been designed with ease of programming in mind. This is reflected by the easy-to-master programming language and by the fast compiling and linking mechanisms.

On the other hand, creating a command still requires some overhead. A new module has to be written, exporting a procedure implementing the command. This is sometimes considered a burden, too heavy for the user. Furthermore, managing module loading and unloading requires some basic knowledge about the system.

Using the same language and environment for programming and scripting allows for a seamless transition between scripting and general programming. Scripts can use all of the programming environment's power. A script can easily be converted into a regular program, allowing for rapid prototyping. The later can also be achieved, if the scripting language is a subset of the programming language.

The Oberon System, for instance, could be extended with an interpreter executing Oberon statement sequences, but not allowing for type or procedure definition.

While the previous discussion suggests to use a general purpose programming language for scripting, today's mathematical computing environments seem to have chosen the opposite approach. The languages defined for their programming exhibit some features of a scripting language. Programs are interpreted. Programming focuses on combining operations, predefined in the language. The languages are rather specialized for the problem domain but not for general programming.

The main consequence of emphasizing simple, mathematics-related programming is that scripting is restricted to the mathematical area. Using a general purpose programming system allows scripts, like any other program, to bridge the gap between different components, where appropriate.

### 4.2.2 Approach 2: Editor Scripting

Many editors for compound documents feature a proprietary scripting component. With OpenDoc, for instance, *OSA* (Open Scripting Architecture) has been defined. Also, some software environments feature a scripting facility, like *AppleScript* of Apple's System 7. Unix shell programs fall into a similar category. Depending on their particular capabilities, these scripting facilities can be used with a document-based user interface.

This kind of scripting focuses on editing operations rather than on data manipulation. In general, the instructions in a script are operations similar to those invoked directly by the user. Many editor scripting components feature a recording facility to automatically generate a script from an editing sequence.

At one extreme, editor scripting can be regarded as an attempt to overcome a bad overall software architecture: the individual software components lack a programming interface. Therefore, the only way to use them in a programmed, non-interactive way is to simulate a (human) user.

### 4.2.3 Approach 3: Interpreting A Document Section

Both methods described so far require the use of tools external to the user interface and workspace implementation. It is preferable to write scripts in the same way as documents. This makes scripting easier, since the same mechanisms are used as in the user interface. Also, such scripts can be used in interactive text books to show a sequence of computation steps. Such scripts require that icons can be included into scripts.

A script is a structure on a higher level than singular objects. It expresses relations between objects. Therefore, the environment of the objects needs to be considered. The environment of an icon is its container. As a consequence, scripts are expressed within containers.

Most naturally, scripts are represented in a container that supports intrinsic data (see Section 2.3.1). Containers supporting text or graphics are particularly well suited. Without loss of generality, we will concentrate on text. Text is much easier to handle than graphics, and textual scripting is better understood than graphical scripting, but the main considerations apply to both.

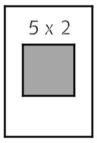
In a text container a script can be represented using a programming language. This language will have one particularity: icons can be used as identifiers.

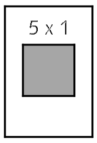
Interchangeability between scripts and programs like with command programming can be achieved by using a subset of the environment's programming language, with the exception of the icons. The scripting mechanism would not support definition of procedures or types. If such is required, the programming environment's language has to be used. The extra overhead for creating a module is relatively small if types or procedures are being defined.

#### Example 9

In one of our projects a document-centered front end for a matrix computation library has been created. This prototype featured a script interpreter. The language used was derived from Matlab [Mat–90]. A script consists of a sequence of statements. Each statement is either an expression or an assignment of an expression to a variable. Assignment statements result in a name binding. Statements not containing an assignment produce an icon as reference to the expression's value. In the particular implementation these icons were pasted to the system's global insertion focus, i.e. the same place where characters typed on the keyboard would occur.

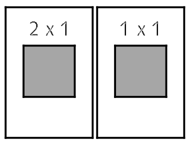
A sample script is given below. First, a matrix and a vector are assigned to the names *A* and *b*. This "input" is stated by two icons. The function *MatLib.QR(A)* computes two matrices, assigned to *Q* and *R*. The fourth line and the seventh line do not contain an assignment, these statements produce an output icon instead.

A := 

b := 

```
Q, R := MatLib.QR(A)
R \ (Q' * b)
c := (Q' * b)
c := c[2:4, 0]
c' * c
```

Interpretation of the above script leads to two icons representing the script's results. The icons are listed in the order in which they are produced. The first icon is produced by the fourth statement, the second icon by the seventh statements of the above script:



This approach has one major disadvantage. A contiguous section of a document to be interpreted as a program must not contain any other text. Consider for instance an interactive text book, in which sequences of operations are explained. If these operations should be interpretable as a script, they must be clearly separated from the explaining text. To achieve that, text must either be written within the syntactical corset of comments, or statements must be marked specially. The latter can be done with icons. An alternative provides the fourth approach, explained below.

### 4.2.4 Approach 4: Interpreting Special Icons

Each statement of the script is represented by an icon. This style of scripting can be seen as a statement oriented form of *Literate Programming* as defined by [Knuth–84a]. The interpreter extracts the executable statements from a document. When creating such a document the needs of readability can be emphasized. The script can be scattered over the document, not affecting its structure. As a result, the document can be more readable than with the previous approach.

#### Example 10

This example illustrates how an interactive text book could look like. The actual computational commands are hidden in the corresponding script parts. The script combines local computing with a remote Maple server.

In the text below, all formulae including an assignment (denoted by ":=") are represented by a special icon. (These are not marked further in the paper version.) The script interpreter ignores everything, except for these icons. During interpretation, for each icon a specific action is executed and the result is displayed by the icon. The actions to be executed are not displayed by the icon. To review or edit them, the icon has to be opened through a mouse click. This allows to create better readable texts. (Otherwise the action descriptions would have to be integrated into the text.)

The following text is shown as it results after evaluating the script.

Let $f(y) := \int_0^y \left(4 \cdot x + 2 \cdot x^3 + 3 \cdot x^2 + 5 + x^4\right) dx$.

Using Maple to integrate we obtain $f(y) := 2 \cdot y^2 + \frac{y^5}{5} + \frac{y^4}{2} + y^3 + 5 \cdot y$.

Substituting 4711 for y we get: $f(4711) := 2 \cdot 4711^2 + \frac{4711^5}{5} + \frac{4711^4}{2} + 4711^3 + 5 \cdot 4711$.

This can be simplified to $f(4711) := \frac{4643.....87}{10}$,

which Maple can factor into $f(4711) := \frac{3 \cdot 7 \cdot 13 \cdot 23 \cdot 53 \cdot 61 \cdot 673 \cdot 339871517}{2 \cdot 5}$.

The script statement associated with the last icon in the script above (performing the factorization), may look as follows:

```
f(4711) := ExprMaple.Eval ifactor(?)

applied to f(4711)
```

### 4.2.5 A Note On Name Bindings Within Scripts

We have stated that name bindings have several shortcomings when used in a dynamic, interactive situation. These shortcomings result from the absence of static scoping. For scripts, static scopes can be defined: each script has its own, completely encapsulated scope. Total encapsulation is then feasible, since the script can communicate with its environment through icons.

From the user's perspective execution of the script interpreter is an atomic action. Name bindings can be used exactly within the scope of a single command invocation, but after the command's termination all name bindings are released.

### 4.2.6 Discussion

The four implementations of scripting discussed have different strengths and weaknesses. Command programming is best suited for programming algorithms and using libraries. Editor scripting is useful to automize sequences of editing operations. Because of its flexibility, interpretation of document sections is the choice for experimenting with simple sequences of mathematical operations. Interpretation of icons finally has its strength in presenting tutoring texts that should be checked for consistency automatically or allow the reader to experiment. It should be noted, that the former two approaches involve mechanisms outside the workspace implementation. The latter two use the same reference mechanism to objects as the user interface.

One common point of strength is the independence from the computational tools used for the computation. This independence results from abstracting all computations by non-interactive commands. Commands using different remote services and commands computing locally can be arbitrarily interleaved. A single script can involve computation of all these kinds. Special consideration has to be given to the fact that remote computations are asynchronous activities (see also Section 4.1.6).

The last approach, interpretation of scripts scattered into several icons, could easily be augmented with an automatic evaluation mechanism. This would implement a constrained editor, since the results visible in the various icons would always satisfy some specified relations. The resulting user interface would be similar to that of Mathcad [MC–93], which resembles a kind of spreadsheet. Unnecessary work (redoing calculations on unchanged data) could be avoided either by using some heuristics or by leaving the decision to the user, when to make the entire document consistent. The latter is exactly what is offered by the implementation discussed above: a command is offered which computes all the script statements on demand.

## 4.3 A Simple, Generic, Expression Editor

Mathematical expressions need to be entered and edited by the user. The original compound document concepts suggest that parts representing expressions also encapsulate a complete expression editor. For this, existing graphical expression editors can be transformed into document parts. Workspaces and scripting can be implemented using such parts. From another point of view one could say that any functionality of an expression editor can be incorporated into an icon. A comprehensive discussion of various techniques for expression editing can be found in [Soiffer–91].

This section describes an implementation of an expression editor based on the interaction of containers and parts in compound documents. It is not the main goal to introduce a new user model, neither to create a particularly "powerful" or "comfortable" expression editor. Instead, we focus on extensibility and light-weightness. Much functionality can be gained with little effort by using the material introduced for generic drawing (Section 3.5) and icons in context (Section 4.1.4). The point to be demonstrated is how much can be achieved with small software when the design is good. The editor may not fulfill everybody's wishes, but it is probably extensible towards them.

An important design goal is extensibility with respect to new expression types. Too many mathematical notations exist to include all of them in a closed editor. The complexity of the editor and its extensibility are related directly: the fewer concepts are used, the fewer an extension programmer must understand and implement. Defining a small interface between the component framework and expression types is the crucial software engineering task. Also, from the user's perspective it is advantageous if the editor is built on a few generic principles. This makes editing uniformly, independent of the type of the expression being edited.

Sections 4.3.1 shows how interpretation of expression icons within text can be used for editing. The resulting text-based editor is very simple and easy to grasp by the user. In a second step (Sections 4.3.2 to 4.3.4) this editor is turned into a light-weight graphical expression editor, featuring standard editing mechanisms.

### 4.3.1 Editing Expressions By Partial Conversion To Text

A compound document container featuring text as intrinsic data allows the combination of textual and graphical presentations of expressions. Whereas text is easier to edit, graphics are easier to understand. To combine these advantages icons are used to achieve an appealing two-dimensional appearance, and text is used for expression editing and entering.

The key is the ability to easily transform expressions between the various representations (pure text, text containing icons, single icon). Most important, this conversion can be done gradually, since an expression can be represented by a mixture of text and icons. This gives access to subexpressions for editing, in contrast to tools allowing for textual representation only of complete expressions. With the latter, the user may loose orientation if the expression is too large.

An in-place editor would depend on the type of the expression represented by it. Editing text in turn is generic. Text has no further semantics, meaning is associated by an interpreting parser.

Roughly, the interface of an expression type abstracts:

- ○ displaying expressions on a two-dimensional plane
- ○ conversion to text
- ○ conversion from text.

Expression drawing and conversion from text are already implemented for workspaces and scripting (see Section 4.1.4.). The collapse (or iconize) command retrieves the current selection and replaces it with a single icon.

Gradual transformation to text can be done along the structure of the generic data graph representing the expression. The root node is converted to text and its subexpressions are represented by individual icons. The result of this operation will be called the *first-level decomposition* of the original expression. The first-level decomposition of an expression representing a polynomial for instance would be a text consisting of several icons (one for each monomial) with plus/minus signs in between.

These two mechanisms are sufficient for expression editing. The part of an expression to be edited is transformed to text by iterative first-level decomposition. Once arrived at the desired subexpression, the text can be changed through text editing operations. Finally, the entire text is converted back to an icon, using the command described above. (See Example 11)

This procedure is independently of expression types. Decomposition follows the generic graph structure. Only conversion of the individual information of a node to text depends on the node's type. For this, a procedure must be bound to the node's type.

Obviously, decomposing an icon is a frequent operation. It should be associated with a mouse click on the icon.

Note that the this example intends to show how the transformation is done manually. Alternatively, a simplification command may be used to perform the same transformation.

The procedure shown involves a minimal number of concepts to be understood by the user. No knowledge is needed apart from how to use the text editor and the conversion mechanisms. However, other possibilities exist and will be explained next. A first improvement is to make selected subexpressions directly accessible.

**Example 11**

To replace the fraction by the corresponding *tan* within

$$\int_0^{\pi} \frac{\sin\left(\frac{x}{2\cdot\pi}\right)}{\cos\left(\frac{x}{2\cdot\pi}\right)}\,dx$$

the expression can be decomposed on its first level by clicking on it with the mouse:

$$\mathrm{int}\left(\frac{\sin\left(\frac{x}{2\cdot\pi}\right)}{\cos\left(\frac{x}{2\cdot\pi}\right)}, x, 0, \pi\right)$$

One further mouse click on the fraction leads to:

$$\mathrm{int}\left((\sin\left(\frac{x}{2\cdot\pi}\right) / \cos\left(\frac{x}{2\cdot\pi}\right)), x, 0, \pi\right)$$

The numerator *(sin(...))* and the slash (indicating the fraction) can be be deleted from the text now:

$$\mathrm{int}((\cos\left(\frac{x}{2\cdot\pi}\right)),\ x,\ 0,\ \pi)$$

After a third decomposition (of *cos(...)*), the expression has been expanded enough to change the function name *cos* to *tan* by plain text editing:

$$\mathrm{int}((\tan(\frac{x}{2\cdot\pi})),\ x,\ 0,\ \pi)$$

Selecting this text and applying the *collapse*-command leads directly to the desired two-dimensional representation:

$$\int_0^{\pi} \tan\left(\frac{x}{2\cdot\pi}\right)\,dx$$

## 4.3.2 Selecting Subexpressions

Editing expressions graphically involves two different mechanisms: pure graphical manipulation using a pointing device and parsing input from the keyboard, providing immediate two-dimensional feedback. Graphical manipulation means to copy and/or replace selected subexpressions. New expressions can be input by composing them from predefined patterns. A particularly flexible, configurable, and light-weight implementation of this uses copy and paste. All these operations are based on selections of subexpressions within icons.

Generic drawing of expressions relies on bounding boxes to abstract the properties common to all expression types (see Section 3.5.).

In detail, a bounding box carries the following information:

- the spatial extension (width, height, horizontal baseline, optionally a vertical baseline)
- a reference to the expression object
- references to the boxes of the immediate subexpressions and their relative locations
- a method to draw the expression onto a (scalable) bitmap.

Bounding boxes allow to select a subexpression with a pointing device. Every point in the X-Y-plane can be associated with the innermost expression visible at this point. Tracking such selections is independent of expression types and can be controlled by the event handler of expression icons.

Expression selections can be integrated with text selections. An expression can be put into any text by generating an icon for it. If a selected text stretch can be interpreted as an expression, such a selection can be pasted as a subexpression. (An attempt to paste text not being valid as an expression will lead to an error message and leave the destination unchanged.) Expression selections and text selections become interchangeable. The copy feature of the text system can be extended onto selections within icons.

Subexpression selections can improve the text-based expression editing discussed in Section 4.3.1. Instead of converting the surrounding expression to text, the subexpression can be copied out for editing. Icons may recognize a special mouse command as a shortcut for such operations. This command opens a view presenting the selected subexpression for editing. This view serves as a scratchpad. An update command allows to replace the originally selected subexpression. Alternatively, every editing operation may invoke the parser leading to an automatic update whenever possible. Example 12 shows how Example 11 can be done with this.

It should be noted, that this approach fits well into the user model of the Oberon System (which has been used for the implementation presented in Chapter 5). Data within a Text Element can usually be edited in a similar way: clicking the mouse on the Text Element opens a separate view presenting a special purpose editor. In the case of expressions this editor is the text editor itself. Furthermore, in the Oberon System views are very light-weight objects. For instance, no separate process is created for them. Therefore, opening an extra view is fast. It is perfectly feasible to base the editor on this.

### Example 12

With the mouse the fraction can be selected within

$$\int_0^{\pi} \frac{\sin\left(\frac{x}{2\cdot\pi}\right)}{\cos\left(\frac{x}{2\cdot\pi}\right)}\,dx.$$

The selection can be opened in a new text view

$$\sin\left(\frac{x}{2\cdot\pi}\right) / \cos\left(\frac{x}{2\cdot\pi}\right).$$

This representation can be changed to $\tan\left(\frac{x}{2\cdot\pi}\right)$ as shown in Example 11. An update command (or automatic update) substitutes this expression for the original fraction.

**Example 13**

Consider that the leading coefficient of the following polynomial shall be changed to 17.

$$3\cdot x^5+25\cdot x^4+94\cdot x^3+210\cdot x^2+271\cdot x+165$$

This can be done by selecting the number *3* with the mouse, opening the selection within a separate view which will contain the text "3", changing this text to "17", and updating the expression icon.

### 4.3.3 Well-Formed Expressions

**Definition:** A mathematical expression, represented in any way, is called *well-formed* if all required subexpressions are present and well-formed and are of the correct type (if any requirements are imposed).

**Examples**

- A fraction is well-formed if both the numerator and denominator exist and are both well-formed.
- A well-formed integral has either no bounds specified or both upper and lower bound are given and well-formed. The subexpression specifying the integration variable may be required to be of type *symbol*.
- Mathematical correctness of an expression is not required. An expression containing a division by zero or an illegal limit can still be well-formed.

With typed expression graphs as introduced in Section 3.1, ill-formed expressions can be excluded. Explicit assertions may be required (and enforced) when converting from text or generating new objects.

Generally, an editor serves the user better if expressions are guaranteed to be well-formed. Mistakes can be detected earlier. Also, computing commands can rely on well-formed input parameters. This helps to improve efficiency and, more importantly, ease of extension. If it is guaranteed that expressions are always well-formed many explicit checks and assertions can be avoided. Also, the type-dependent part of expression drawing can be kept simpler.

#### Operations On Well-Defined Expressions

Insertion or deletion operations are not allowed when editing well-formed expressions (in contrast to editing text or graphics). In general, deleting a part from an expression or adding a new one can lead to an ill-formed expression. To cope with this, special treatment depending on the involved expression's types would be needed. If possible, such special treatment should be avoided to keep the editor as simple and easily extensible.

Consequently, only a single editing operation must be supported on expressions: replacement of a subexpression. This editing operation corresponds to the typical operation on expression graphs described in Section 3.3. As a further consequence, the insertion focus within an expression is always a subexpression selection. This is a rather drastic approach, but it simplifies the editor considerably.

Incomplete expressions must be represented and edited in text form. However, such incomplete expressions can be avoided by using pattern-based input.

### 4.3.4 Patterns, Templates, Overlays, And Palettes

Expressions can be input graphically using *patterns*. Such patterns are presented to the user within *palettes*. Depending on the way they are used, patterns can be divided into *templates* and *overlays*.

A *template* is a pattern to be embedded into the target expression. It is picked from the palette and copied as is. An *overlay* contains a special, unfilled slot to be filled by the expression to be replaced, i.e. the overlay is "wrapped around" the selected expression (see Example 14).

The conceptual issues of templates and overlays are discussed in depth in [Soiffer–91]. The differences between templates and overlays become apparent when we seek a natural way to input expressions. With templates, an expression is input in prefix style, whereas overlays lead to infix and postfix styles. To enter $a+f(x)$, the following sequence of templates has to be applied: $?+?$, $a$, $f(?)$, $x$. The same expression is generated by applying the following sequence of overlays: *a*, *?+?*, *x*, *f*(*?*). The most natural sequence requires to use both templates and overlays. If *?+?* is applied as an overlay and *f*(*?*) is applied as a template, the sample expression can be entered as it would be read: *a*, *?+?*, *f*(*?*), *x*.

It can be concluded that an extensible editor should feature both mechanisms. The choice of which to apply should be left to the user as it may depend on the type of the concrete expression.

To pick a template from a palette is the same as copying a selected expression. Hence, templates do not require a specific implementation. The copy mechanism can be used, and any document containing expressions can serve as a palette. The resulting advantage is the same flexibility as introduced into the Oberon System by Tool Texts, replacing traditional menus: instead of configuring palettes through a separate mechanism, they are generated, changed, and stored like any document.

Ease of use of this template implementation depends heavily on how copying is presented to the user. Oberon's *copy-over* mechanism is particularly handy for this purpose. It allows to select source and target in either order and to issue the copy command simply by an additional mouse click while tracking the second selection. Therefore, copying a template requires a single extra (inter-) click, compared to a specialized palette implementation.

With other models for copying, like *copy & paste* which uses two separate operations to copy to and from a clipboard, the lean template implementation would be too cumbersome to use. It has not yet been investigated, how good *drag & drop* would suit the purpose.

To allow entering an expression of a new type, the software component defining this type needs to be loaded. This happens automatically when a document (maybe a palette) containing an expression of the type in question is opened, since it is the task of the document editor to load the code required by the parts within a document. Hence, the pattern-based extensible expression editor is being configured by opening palettes. Alternatively, a component can be loaded explicitly through a command.

*Overlays* can be implemented with the help of a small extension: a special expression type *placeholder*. Upon replacing a subexpression, every placeholder is substituted by the originally selected target expression, which otherwise would have been discarded. Thus, the pattern copied is being wrapped around the selected expression. Obviously, a pattern containing no placeholder is simply replacing the target, i.e. it is treated as template. Hence, overlays and templates are distinguished by the occurrence of placeholders within them.

Also in Kaava [Rimey–92] placeholders are used to distinguish templates from overlays.

[Soiffer–91] suggests that both templates and overlays could also be entered through the keyboard as an alternative. This requires assignment of patterns to function keys or special key macros. In an extensible environment such assignments have to be created as configurations. Whereas palettes are visible, assignments to sequences of key strokes are invisible and must be memorized by the user. This is an advantage for expert users, but a disadvantage for others.

On the other hand, it is easy to implement such an extension. For configuration, a special document could be introduced, containing icons representing the patterns and the corresponding key-codes. However, this functionality does neither present a conceptual invention nor is it particularly difficult to realize. Thus, it has not been implemented in the prototype.

Many expression editors include *undefined symbols* into their templates. These are to be filled with concrete subexpressions later. Here, undefined symbols are a superfluous addition, since every subexpression can be selected for replacement. Still, undefined symbols could be used to identify positions where further input is required. With this information selections could be made automatically, allowing the user to enter subexpressions continuously without selecting the input focus each time. This concept could be added to expression icons without affecting the overall design. Therefore, it has been omited from the prototype implementation.

**Example 14, Templates and Overlays**

Patterns like the following can be used as *templates*:

$$\sqrt[n]{Y} \qquad \|Y\| \qquad \lim_{x \to 0} Y \qquad \lim_{x \to 0} \int_{-\infty}^{x} f(t)\, dt$$

Patterns like the following act as *overlays* when copied to a selection within an expression icon:

$$\sqrt[n]{?} \qquad \|?\| \qquad \lim_{x \to 0} ? \qquad \lim_{x \to 0} \int_{-\infty}^{x} ?\, dt$$

If the template $\sqrt[n]{Y}$ is copied to the integral within

$$\lim_{a \to \infty} \int_{0}^{a} f(x)\, dx$$

the result is

$$\lim_{a \to \infty} \sqrt[n]{Y}$$

If the overlay $\sqrt[n]{?}$ is copied to the same place, the result is

$$\lim_{a \to \infty} \sqrt[n]{\int_0^a f(x)\,dx}$$

### 4.3.5 Parsing

A typed text can be parsed and translated into an expression, as an alternative to graphical input. Because the set of expressions should be extensible, the parser needs to be extensible, too. Hence, syntax-driven editors are often used for this purpose. To extend the editor the language syntax must be extended. Examples of syntax directed editors for extensible languages can be found in the realm of program editors (e.g. Orm [KLMM–94]). Unfortunately, requiring maintenance of a separate syntax description contradicts the design philosophy which has been followed so far in this thesis. This philosophy is that only a single module needs to be programmed and loaded to extend the system with a new expression type. With a syntax driven editor the syntax would have to be changed in addition to programming the module.

While expressions are entered through the keyboard, intermediate states occur where the input is incomplete. Incomplete input can be syntactically incorrect, or represent an ill-formed expression. Two approaches exist to cope with such situations: either parsing and feedback can be delayed until the input represents a well-formed expression or the input can be completed automatically after each change. Automatic completion of ill-formed expressions with some default objects seems to be favored by the present designers of expression editors. Unfortunately, it presents various problems when a light-weight, extensible system is the goal. How to complete an expression depends heavily on the expression's type. In a non extensible editor all the knowledge about how to complete each individual expression can be built in. An extensible editor requires these properties to be specified for each extension.

The easiest way to solve this problem is to give up automatic completion. Expressions can be edited in textual representation using the functionality presented in Section 4.3.1. The text is entirely visible and editable with the well-known and general text editing commands. When the user finishes editing a single mouse click will parse the input.

If this suggestion is considered too primitive, an extension can be offered: when a scratchpad viewer is used as described in Section 4.3.2, the corresponding expression icon can be treated like a second view. Any change in the scratchpad's text leads to an attempt to parse it as an expression. If this attempt is successful, the icon is updated otherwise it is left unchanged. This creates a maximum of feedback to the user: the icon gives two-dimensional, up-to-date feedback whenever possible and the text gives exact information about what has been typed. Unfortunately, after a parenthesis is opened the two-dimensional feedback will not be updated until the corresponding closing parenthesis is entered. This might be a problem with large expressions within parentheses. To resolve this, the parser could try to add the missing parenthesis.

As an additional extension, such a scratchpad viewer can be opened immediately when a character is typed while an icon holding a selection is the input focus. The text in the scratchpad is then initialized to hold only the character typed, and the scratchpad receives input focus. Later, when the *Return*-key is pressed, the scratchpad viewer can be closed. This allows to "type into an expression icon" directly. The change done in Example 13 above, can be done by the following sequence of key strokes, after selecting the *3* within the icon: *1*, *7*, *Return*.

A further alternative may be to omit the extra viewer and keep the text hidden. Though this would avoid the user's distraction from the graphically presented expression (to the extra viewer), it gives no feedback about the actual typed sequence of characters. Furthermore, editing this sequence is less direct. It is not possible to position the insertion mark at any point, selections cannot be used, etc. Overall, introducing this hidden state is not justified by avoiding the additional viewer.

### 4.3.6 Discussion

Two different approaches to expression editing have been made available: conversion of relevant parts to text and direct graphical manipulation. Both are implemented generically, allowing for extension with new expression types. Extensions are provided by new software components, presented as single modules including everything needed. No separate configuration files need to be extended, as for instance with CAS/PI [Kajler–92] where the graphical appearance needs to be specified in separate files using a different language (called PPML).

Most of the functionality needed for editing must already be implemented for workspaces and scripting. The interfaces of the expression objects need not be further extended. Also, bounding boxes are needed as generic abstractions of expressions for drawing. These are a sufficient basis for graphical operations. Because of this, and because much of the functionality can be taken from the underlying compound text document framework, the editor is very light-weight.

It should be easy to learn how to use the text-based editor: only primitive operations for conversion between text and icons have to be learned in addition to the already known text operations. Also, the copy-over concept is well known to users of the text system. Thus, templates and overlays appear as natural ways to use the editor.

Using the copy mechanism of the underlying framework leads to an extremely flexible implementation of palettes with templates and overlays. Any expression visible anywhere on screen can serve as a pattern for a template or an overlay. Palettes with such patterns are normal documents and can be handled as such: they can be stored in files, edited, printed, etc. Palettes are easy to modify. They can be tailored for individual users or even for individual projects. Generation of specialized patterns of arbitrary complexity is affordable. There is no strict limit between such palettes and a scratchpad. In general, this flexibility corresponds to that of the *Tool Texts* within the Oberon System [Reiser–91].

Templates can also be used in combination with text editing: one can copy a template and modify it in its textual representation. This is useful, since in some cases it can be faster than mouse driven, graphical editing. The textual template is a pattern for the user's input. Seen together with its graphical appearance it is also self-documenting: equally named subexpression can be identified and their meaning can be derived from the graphical appearance.

To enter an integral, for instance, the following template can be used:

$$\int_a^b f(x)\,dx$$

For editing it can be converted to textual form:

int(f(x), x, a, b)

The meaning of the parameters *f*(*x*), *x*, *a*, and *b* are obvious from comparison with the graphical representation.

It is considered a particular strength of the editor that all the editing techniques available can be combined by the user on a very fine grain level. This allows the creative user to invent editing methods that have not been anticipated by the designer of the software.

The editor lacks some functionality which may be wanted by particular users (e.g. automatic selection or entering templates through the keyboard). In principle, this functionality could be implemented within icons. Due to the modularity of the component-oriented implementation such changes require to replace the icon component only. They do not affect the rest of the framework. However, in the author's experience the implemented editor turned out to be very practical. Apart from occasional computations with Maple it has been used to write this thesis.

# 5 Implementation

This section describes an implementation of the concepts presented in Chapters 3 and 4. A complete framework for mathematical software has been implemented. It includes abstract expression graphs, expression displaying within icons, a text parser, and an expression editor. To prove its extensibility the framework has been extended with a set of expression types, a rational number simplification command, and a connection to a Maple server. The latter uses a communication link.

The environment used for the implementation is the Oberon System [Reiser–91, WiGu–89, WiGu–92]. It is programmed using the programming language Oberon-2 [Mössenböck–91], an extension of the language Oberon [Wirth–88]. The latter is a successor of Modula-2 [Wirth–82] and hence of Pascal and Algol. The extensible text system described in [Szyperski–92] has been used as a compound document framework. Icons are implemented as Text Elements. Components are Oberon Modules.

The implementation serves as a proof of concept, but also has value in its own. It presents a working extension of the Oberon System allowing to edit expressions, include them into documents (like this thesis), and perform mathematical operations on them. It gets most of its mathematical power from a link to Maple. The latter proves that the editor-based user interface works also in a client/server architecture.

The framework has been divided into several components, i.e. modules. The basic module (Module *Expressions*) implements immutable expression graphs as an abstract and generic data structure. According to the Model-View-Controller Separation (MVC) the visual presentation is abstracted within a separate module (Module *ExprViews*). These two modules are used by a third one that implements expression icons as Text Elements and allows to embed expressions into texts (Module *ExprIcons*).

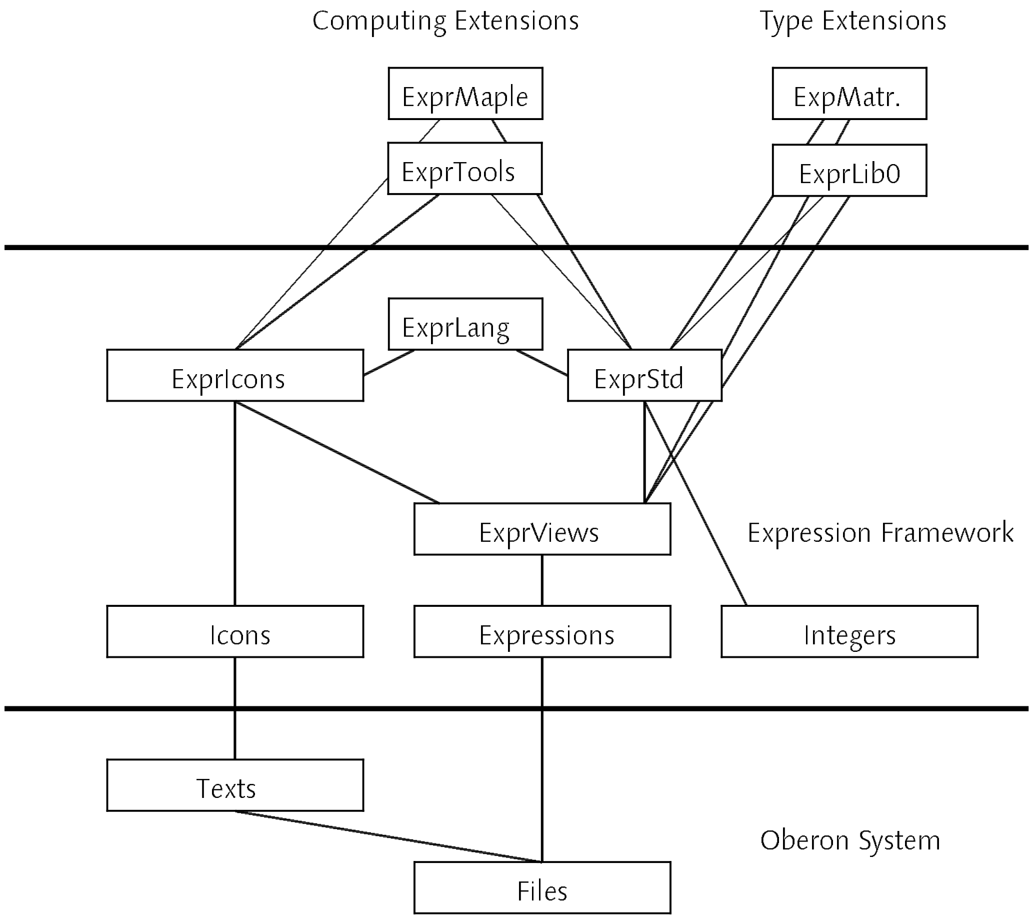


*Figure 2: the modules of the expressions framework in Oberon*

Another, independent module (Module *ExprStd*) defines basic expression types (integers, symbols, binary operators, functions, collections, indices). These are used to define a primitive language for textual presentation of expressions. Conversions between the standard expression types and text are done by a module, implementing the parser (Module *ExprLang*). Figure 2 shows the modularization of the framework and Table 2 lists all modules and their size in statements.

A variety of sample extensions has been implemented, too. We must distinguish between modules contributing new expression types and modules implementing computational algorithms. Examples of new expression types are special constants (π, ∞, greek symbols, etc.), roots, integrals, binomial coefficients, and others. A general substitution command has been implemented as an example for the typical substitution operation on expression graphs (see Section 3.3). A simplifier for rational numbers has also been implemented. The Maple link is a further extension.

| Module | Purpose | Statements |
|---|---|---|
| Integers | arbitrary precision integers | 703 |
| Icons | general icons within text | 518 |
| Expressions | generic expression graphs | 422 |
| ExprViews | basis for expression drawing | 448 |
| ExprStd | standard expression types | 639 |
| ExprLib0 | more expression types | 1185 |
| ExprMatrices | expression type for matrices | 199 |
| ExprBessel | expression type for Bessel functions | 103 |
| ExprIcons | expression icons | 663 |
| ExprLang | generic parser | 363 |
| ExprTools | general command toolbox | 305 |
| ExprRatCalc | rational number simplifier | 268 |
| ExprSubstituter | example substitution command | 50 |
| ExprEvalIcons | evaluatable icons for scripting | 582 |
| ExprMaple | link to Maple | 272 |
| | | |
| Total | | 6720 |

*Table 2: number of statements per module*

In the following (i.e. in Sections 5.1 to 5.3) several abstractions defined by the framework are described. To illustrate their usage the relevant parts of the implementation of the expression type *Power* are shown.

## 5.1 Immutable Directed Graphs

### 5.1.1 A Design Pattern Based On Carrier-Rider-Separation

Chapter 3 states that expressions shall be represented by immutable directed graphs. It is challenging to design and implement an extensible abstract data type that enforces immutability. Part of the challenge is not to hinder ease of graph creation too much. The interface shall provide an easy to understand and easy to use abstraction. The typical substitution operation stated in Section 3.3 should be kept in mind.

Symbolic computation systems like Maple [CGGLMW–91] use immutable DAG structures, too. For protection, a higher level is introduced, the Maple language. The language interpreter hides critical parts of the lower level. Immutability is not guaranteed for programs that are at the same programming level as the data structure is implemented in.

The problem of implementing immutable graphs can be reduced to implementing immutable lists. Each node in the graph refers (immutably) to such a list. The list contains the ordered successors of that node.

From the user's point of view it would be ideal to allow arbitrary list manipulations. This is possible, as long as one restriction is obeyed: whenever a list can be referred to by a component extending the basic system, the list must be frozen. Attempts to change such a list must generate a copy which can be manipulated freely as long as no further reference to it exists. Hence, the key is to control when a reference is passed outside the basic component for the first time.

How can an extension component manipulate a list without holding a reference to it? A solution is provided by *Carrier-Rider separation*. It originates in the Oberon system [WiGu–92] where it has been invented as an access mechanism to sequential structures, like files and texts. Its purpose has been to separate the access-related information (Rider) from the data structure (Carrier). For instance, the current reading position is stored in the Rider. This allows for multiple objects to access the same Carrier at different positions simultaneously. Each object simply uses a separate Rider connected to the common Carrier. Later, Carrier-Rider separation has been generalized in [Szyperski–92b] to allow for independent extensions in the domains of Carriers and Riders.

To implement immutable lists, the Rider is used to hide a reference to a list. A direct reference to the list can be obtained through a method of the Rider only. This method marks the list as frozen. To change a frozen list it must be copied first. The copy can be changed freely, until it is frozen, too. Copying is done on demand by the Rider's change methods. To the user of this interface, copy on demand and freezing is entirely invisible.

Immutable graphs implemented with immutable lists of successors can easily be guaranteed to be DAGs. Cycles can be excluded by a single, easy to assert requirement: each node to be inserted into a list needs to be complete, i.e. a list of successors must have already been assigned to it. Consequently, a completed node can only refer to other already completed nodes. This applies transitively. Hence, it is not possible to build a cycle. This guarantee is an advantage in our context.

The above design pattern, has been derived from the general Carrier-Rider pattern described in [Szyperski–92b]. It additionally employs information hiding to guarantee immutability of the Carrier.

This design pattern relies on information hiding at a level different than single objects. Modules as discussed in [Szyperski–92a] serve the purpose in an excellent way. The base module of the expression framework asserts the necessary invariants between Riders and Carriers, both defined within it. Only such variables are exported, whose change cannot affect the invariants.

### 5.1.2 Excerpt From The Interface Of Module Expressions

```
TYPE
   Message = RECORD END;                              (* base type for messages *)

   List = POINTER TO ListDesc;                        (* immutable lists of objects *)

   Expression = POINTER TO ExpressionDesc;
   ExpressionDesc = RECORD                            (* abstract expression type *)
      successors–: List                               (* list of subexpressions *)
   END;

   Rider = RECORD                                     (* Rider for List access *)
      pos–: LONGINT;                                  (* current Position *)
      exp–: Expression;                               (* current Expression *)
      eol–: BOOLEAN                                   (* End Of List *)
   END;

VAR
   emptyList–: List;

PROCEDURE LengthOf(l: List): LONGINT;                 (* length of list l *)
PROCEDURE Excerpt(of: List; beg, end: LONGINT): List; (* copy of [beg ... end[ *)

PROCEDURE OpenRider(VAR r: Rider; l: List);           (* open r on l; position at 0 *)
PROCEDURE Set(VAR r: Rider; pos: LONGINT);            (* reposition open Rider *)
PROCEDURE Forward(VAR r: Rider);                      (* forward r by 1 *)
PROCEDURE Change(VAR r: Rider; exp: Expression);      (* change expression at r.pos *)
PROCEDURE Insert(VAR r: Rider; exp: Expression);      (* insert entry at r.pos; forward r *)
PROCEDURE Delete(VAR r: Rider; len: LONGINT);         (* remove [r.pos ... r.pos + len[ *)
PROCEDURE ThisList(VAR r: Rider): List;               (* freeze list, do not change r *)

PROCEDURE Init(e: Expression; successors: List);      (* initialization, call only once *)
```

Three types are involved so far: *Expression*, *List* of successors (i.e. subexpressions), and *Rider* on such a list. Only the type *Expression* will be extended within other components. With every object of type *Expression* a list of successors is stored. This must be assigned using the procedure *Init*, which must be called exactly once for every expression. An expression is initialized properly only if its list of successors does not have the value *NIL*. An expression not referring to any subexpression must have the empty list assigned. This property can be used by Procedure *Init* to prohibit an object from being initialized twice.

Types *List* and *Rider* are implemented entirely within Module *Expressions* and are not meant to be extended. According to the design pattern, lists can be manipulated through *Rider*s only. Therefore, the implementation details of Type *List* are not exported. It is only allowed to determine the length of a list (Procedure *LengthOf*) and to copy out a sublist (Procedure *Excerpt*).

Riders can be opened and positioned on lists. They allow to change, insert, or delete elements. A reference to an immutable list can be obtained from a Rider (Procedure *ThisList*).

A global, read-only exported variable makes an empty list available. It is used as the starting point to create new lists.

An important part of the implementation is hidden within the module. Objects of type *List* refer to linked lists. Additionally, a boolean variable specifies whether the list is frozen, i.e. whether a reference to the list has been passed outside the module yet. This information is changed by a Rider. Riders refer to the list they operate on, their current position on that list, and the corresponding entry.

### 5.1.3 Excerpt From The Implementation Of Module Expressions

```
TYPE
    Link = POINTER TO LinkDesc;
    LinkDesc = RECORD
        next: Link;
        exp: Expression
    END;

    Rider = RECORD
        pos-: LONGINT;                  (* current Position *)
        exp-: Expression;               (* current Expression *)
        eol-: BOOLEAN;                  (* End Of List *)
        link: Link;                     (* current link in list, holding exp *)
        list: List                      (* head of current list *)
    END;

    ListDesc = RECORD
        links: Link;                    (* first entry *)
        len: LONGINT;                   (* number of entries *)
        frozen: BOOLEAN                 (* = list must not be changed *)
    END;


PROCEDURE OpenRider(VAR r: Rider; l: List);     (* open r on l; position at 0 *)
BEGIN
    r.list := l; r.pos := 0; r.link := l.links;
    IF l.len # 0 THEN r.exp := r.link.exp; r.eol := FALSE
    ELSE r.exp := NIL; r.eol := TRUE
    END
END OpenRider;
```

```
PROCEDURE ThisList(VAR r: Rider): List;             (* freeze list, do not change r *)
BEGIN
    r.list.frozen := TRUE; RETURN r.list
END ThisList;

PROCEDURE Change(VAR r: Rider; exp: Expression);     (* change expression at r.pos *)
BEGIN
    ASSERT((r.pos < r.list.len) & (exp # NIL));
    IF r.list.frozen THEN CopyList(r) END;           (* copy on demand *)
    r.link.exp := exp; r.exp := exp                  (* change entrie's value *)
END Change;

PROCEDURE Init(e: Expression; successors: List);
BEGIN
    ASSERT(e.successors = NIL);                      (* e is not yet been initialized *)
    ASSERT(successors # NIL);                        (* parameters are valid *)
    e.successors := successors                       (* set values *)
END Init;
```

### 5.1.4 Generating An Expression (Example)

The following example shows how to generate a node of an expression graph using the interface defined above. Section 5.4 describes a substitution command as a more elaborate example of expression manipulation, involving data changing and replicating graph parts.

The expression type *Power* is defined below. It is an extension of the abstract type *Expressions.Expression*. It contains two additional fields to provide direct access to the two subexpressions (base and exponent). Allowing to access subexpressions directly via record fields is a pure optimization. Algorithms being aware of the type *Power* can use these record fields instead of setting up a Rider as required for generic programming. The implementation of Type *Power* has to assert the consistency of the record fields with the successors list.

```
TYPE
    Power* = POINTER TO RECORD(Expressions.ExpressionDesc)
        base-, exponent-: Expressions.Expression     (* for direct subexpression access *)
    END;

PROCEDURE NewPower*(base, exponent: Expressions.Expression): Power;
    VAR p: Power; r: Expressions.Rider;
BEGIN
    ASSERT((base # NIL) & (exponent # NIL));
    Expressions.OpenRider(r, Expressions.emptyList);  (* generate successors list *)
    Expressions.Insert(r, base); Expressions.Insert(r, exponent);
    NEW(p); Expressions.Init(p, Expressions.ThisList(r));
    p.base := base; p.exponent := exponent;           (* set fields for direct access *)
    RETURN p
END NewPower;
```

Procedure *NewPower* allocates and initializes a new object. The two subexpressions are passed as parameters. Execution of the procedure will generate a new successor list: a Rider is set on the empty list and the new items are inserted. The new list is assigned to the new expression object through Procedure *Expressions.Init*.

## 5.2 Properties Of Expressions And Operations On Expressions

Additional services and definitions for all expression types need to be provided by the base module. This section discusses what has been omited in Section 5.1.

### Binding Power

Section 3.5.2 states that each expression object needs to provide information allowing to decide whether parentheses have to be put around it. This information is called the expression's *binding power*. We use the term *binding power* instead of *operator precedence* since the concept applies not only to operators but to expressions in general.

The binding power is a constant value for each object. Therefore, it is stored in a record field exported for reading only. The value must be assigned through Procedure *Init*. (The parameter list of Procedure *Init* is extended accordingly.) In accordance with Section 3.5.2, predefined values for binding power are exported as constants. Procedure *Init* asserts that no other values are used.

### Attributes

In some cases expressions need to associate additional attributes with their subexpressions. As an example consider associative, commutative, binary operators which also have an inverse (like +). Such operations and operands can be arbitrarily reordered. Thus, a single object shall represent any number of such operations. For instance, $a-b+c$ should be represented as one expression referring to three successors.

This unified structure is a prerequisite for effective computation on the data, e.g. for generic canonical normal form computation. The successors of a single expressions can easily be reordered. Changing the topology of the graph is more difficult.

Furthermore, expression graphs are not only used as a basis for computation but also for expression representation. Therefore, their structure must be able to reflect what the user actually entered. Expressions within documents should not be forced to a normal form. For example, it must be possible to represent both $a-b+c$ and $(a-b)+c$.

For each subexpression, it must be stored which operation to apply, plain or inverse. In the example that is whether an operand should be added or should be subtracted. This information cannot be stored in the operand's object itself since the latter may also be used in another expression. The information must be stored with the higher level node, i.e. with the operator.

Such information needs to be organized in a list of the same length as the successors list. To avoid duplication of the list organization, the successors list allows to store a single attribute (an integer number) with each entry. The semantics of that attribute depends on the expression the list is assigned to. (The binary operation for adding and subtracting, for instance, defines two values indicating the *plain operation* or the *inverse operation*.)

Attributes are read, set, and changed together with the subexpressions they belong to. Types *Rider* and *Link* mentioned in Section 5.1 are extended accordingly and so are the signatures of Procedures *Change* and *Insert* (see below).

### Canonical Normal Form And Redundant Object Elimination

According to Section 3.4, generating canonical normal forms involves assignment of a hash key to every expression object. Since objects are never changed, this hash key can be assigned when the object is created.

An effective key must be based on three entities: the list of subexpressions, the expression type, and the extra data of the expression object. Hash keys on the former two can be computed generically, only the third value must be provided by the object. A key must reflect the representation of the expression, since the representation of an expression included in a document must not be changed. Hence, no provision for commutative subexpressions is needed.

These observations lead to the following implementation: the key is computed by Procedure *Init* and assigned to a read-only exported record field of *Expression*. The type's name and the expressions and attributes stored in the successors list are hashed by Procedure *Init*. The result is combined with a modifying key, specified as a parameter. This modifier is to be computed by the expression object from its individual data. (Objects without such data can use any constant value, e.g. 0.)

In Section 3.4.4 two possibilities are discussed to deal with the extra state needed to compute canonical normal forms or to eliminate redundant objects. Either a global table can be used, or a table is allocated on demand for a particular computation. As Section 3.4.4 states, a global table must be treated specially by the system's garbage collection. Otherwise, the references in the table would prevent any registered object from being collected. The implementation discussed here uses temporary tables. One reason behind this is that the Oberon System does not allow extension of the garbage collector as necessary. (Some individual implementations of Oberon provide extension hooks for this purpose, but using them leads to non-portable software.)

The abstract data type *UnificationScope* is exported by Module *Expressions*. Such a new table, or scope, can be allocated through the function procedure *NewUnificationScope*. The table must be specified as a parameter when calling one of the normalization-related operations: *Normalize*, *Unify*, *OrderedList*. *Normalize* generates a canonical normal form. *Unify* eliminates references to equal objects from a graph. For the latter, expressions are considered equal if they have the same representation. (See Section 3.4 for a description of the algorithms used.) *OrderedList* is a library procedure to be used by the individual objects, when generating a normalized representation of themselves. It generates a list ordered according to the key and table values.

### Messages And Message Handler

Functionality is bound to expression objects in the form of a *message handler* procedure. Every object refers to such a procedure through a variable of type *Handler*. Messages are records whose type extends a basic type *Message*. When a "message is sent to an object", the respective handler is called with the message as a parameter. The handler's activity depends on the message's type. A handler may ignore messages of unknown type.

This programming methodology based on message handlers is one of the fundamental concepts used in the Oberon system library. For a further discussion of this see [WiGu–92].

Some concrete message types are already defined within Module *Expressions*: The *CloneMsg* is used to retrieve a cloned object with a modified successor list, e.g. for the typical substitution operation on expression graphs described in Section 3.3. The *TestMsg* is used to determine the equality of an object with another one. The *IdentifyMsg* and the *FileMsg* are used for internalization and externalization from and to files.

### Internalization And Externalization

To externalize a DAG into a file it must be linearized. Linearization must take care of multiple references to nodes. Transforming the expression simply to a textual representation (or an abstract syntax tree) would be inefficient. The

DAG would be treated as a tree and data would be replicated in the file.

A DAG can be linearized using an auxiliary table. Every node written to the file is registered in that table. The table is searched for every node prior to writing it. If the node has been written before, it is sufficient to write the index of the table entry. Note that the table itself does not need to be stored with the data. It can be rebuild during internalization. Nodes can be compared by their references. The table stores these references only.

So far, the graph's edges were treated. Let us now consider the graph's nodes. There are two concerns: the nodes type and the type-dependent information stored with it.

The second concern is easy to handle: a message (*FileMsg*) is sent to the node requesting to store or load its data. To refer to the file, the message contains a file Rider. One could either use two different message types to request storing/loading, or a single message type with a boolean field, as done in the current implementation.

Prior to requesting an object to load data from a file, the object must be allocated and the handler procedure must be instantiated appropriately. Two approaches are possible: a meta programming facility can be used, or the name of a generator is stored with the data.

Apart from others, meta programming allows to determine the name of an object's type, and to allocate a new object of a type that is specified by its name at run-time. Meta programming for Oberon is described in [Templ–94]. It is not being used in the current implementation due to its limited availability.

The implementation is based on the same concept which has already been used for the extensible text system. It is described in [Szyperski–92]. Generators are implemented as commands. During externalization the command's name is retrieved from the object by sending it an identification request message (*IdentifyMsg*). The object's message handler assigns values to the module and procedure name fields of that message. These names are externalized with the data.

During internalization the command's name is read from the file and the command is activated. It is expected to allocate an object of the proper type and to deposit it in the base module together with the appropriate message handler procedure (by calling *Expressions.Deposit*). The latter also initializes the object. *Expressions.Init* must not be called for objects initialized through *Expressions.Deposit*. The data otherwise computed and assigned by *Expressions.Init* is retrieved from the file in this case.

### Procedure Init

The values of the read-only variables associated with expressions (handler, binding, and key modifier) are being assigned through Procedure *Init* together with the list of successors. The parameter list of *Init* is extended accordingly. *Init* asserts that only the predefined values are used for the binding power.

## 5.2.1 The Complete Interface Of Module Expressions

```
DEFINITION Expressions;

IMPORT Files;


CONST                                                   (* binding power values *)
  AtomBind = 4; PowerBind = 3; ProdBind = 2; SumBind = 1; NoBind = 0;


TYPE
  Message = RECORD END;                                 (* base type for messages *)

  List = POINTER TO ListDesc;                           (* immutable lists of objects *)

  Expression = POINTER TO ExpressionDesc;

  Handler = PROCEDURE(e: Expression; VAR msg: Message);

  ExpressionDesc = RECORD                               (* abstract expression type *)
    handle–: Handler;                                   (* message handler *)
    binding–: SHORTINT;                                 (* binding power *)
    key–: LONGINT;                                      (* hash key *)
    successors–: List                                   (* list of subexpressions *)
  END;

  Rider = RECORD                                        (* Rider for List access *)
    pos–: LONGINT;                                      (* current Position *)
    exp–: Expression;                                   (* current Expression *)
    attr–: LONGINT;                                     (* attribute of current expression *)
    eol–: BOOLEAN                                       (* End Of List *)
  END;

  UnificationScope = POINTER TO RECORD END;             (* table for normalization *)

  FileMsg = RECORD(Message)                             (* message for loading/storing *)
    store: BOOLEAN;
    r: Files.Rider
  END;

  IdentifyMsg = RECORD(Message)                         (* retrieve allocation command *)
    mod, proc: ARRAY 32 OF CHAR
  END;
```

```
    CloneMsg = RECORD(Message)                         (* get node with new successors *)
        clone: Expression;
        successors: List;
        scope: UnificationScope;
        normalize: BOOLEAN
    END;

    TestMsg = RECORD(Message)                          (* to perform an equality test *)
        with: Expression;
        equal: BOOLEAN
    END;


VAR
    emptyList–: List;


PROCEDURE LengthOf(l: List): LONGINT;                  (* length of list l *)
PROCEDURE Excerpt(of: List; beg, end: LONGINT): List;  (* copy of [beg ... end[ *)

PROCEDURE OpenRider(VAR r: Rider; l: List);            (* open r on l; position at 0 *)
PROCEDURE Set(VAR r: Rider; pos: LONGINT);             (* reposition open Rider *)
PROCEDURE Forward(VAR r: Rider);                       (* forward r by 1 *)
PROCEDURE Change(VAR r: Rider; exp: Expression; attr: LONGINT);
                                                       (* change expression at r.pos *)

PROCEDURE Insert(VAR r: Rider; exp: Expression; attr: LONGINT);
                                                       (* insert at r.pos; forward r by 1 *)

PROCEDURE Delete(VAR r: Rider; len: LONGINT);          (* remove [r.pos ... r.pos + len[ *)
PROCEDURE ThisList(VAR r: Rider): List;                (* freeze list, do not change r *)


PROCEDURE CloneOf(e: Expression; successors: List): Expression;
                                                       (* node with new successors *)

PROCEDURE NewUnificationScope(): UnificationScope;     (* new table for normalization *)
PROCEDURE Unify(VAR exp: Expression; s: UnificationScope);
PROCEDURE Normalize(VAR exp: Expression; s: UnificationScope);

PROCEDURE Equal(x, y: Expression): BOOLEAN;            (* test for equality *)
PROCEDURE EqualLists(l1, l2: List): BOOLEAN;           (* compare two lists *)
PROCEDURE OrderedList(l: List; s: UnificationScope): List;
                                                       (* canonical form of list *)

PROCEDURE Store(VAR r: Files.Rider; e: Expression);    (* store expression graph *)
PROCEDURE Load(VAR r: Files.Rider; VAR e: Expression); (* load expression graph *)
PROCEDURE Deposit(e: Expression; h: Handler);          (* to be called by allocators *)

PROCEDURE Init(e: Expression; h: Handler; bind: SHORTINT; keyMod: INTEGER; succ: List);

END Expressions.
```

### 5.2.2 Complete Implementation Of The Expression Type Power

In the following the complete implementation of the expression type *Power* (introduced in Section 5.1) is given.

```
TYPE
    Power* = POINTER TO RECORD(Expressions.ExpressionDesc)
        base-, exponent-: Expressions.Expression          (* for direct subexpression access *)
    END;


PROCEDURE PowerHandler(e: Expressions.Expression; VAR m: Expressions.Message);
    VAR self, copy: Power; r: Expressions.Rider;
BEGIN
    self := e(Power);
    WITH m: Expressions.IdentifyMsg DO                    (* specify allocation procedure *)
        m.mod := "ExprStd"; m.proc := "AllocPower"
    | m: Expressions.FileMsg DO
        IF m.store THEN (* no proprietary data needs to be written in this example *)
        ELSE
            (* no proprietary data needs to be read in this example *)
            Expressions.OpenRider(r, self.successors);    (* set fields for direct access *)
            self.base := r.exp; Expressions.Forward(r); self.exponent := r.exp
        END
    | m: Expressions.CloneMsg DO                          (* new object with m.successors *)
        NEW(copy); Expressions.Init(copy, self.handle, self.binding, 0, m.successors);
        Expressions.OpenRider(r, m.successors);           (* set fields for direct access *)
        copy.base := r.exp; Expressions.Forward(r); copy.exponent := r.exp;
        (* no proprietary data needs to be copied in this example *)
        m.clone := copy
    | m: Expressions.TestMsg DO                           (* equality test: m.with = self? *)
        m.equal :=
            (m.with IS Power) & Expressions.EqualLists(m.with.successors, self.successors)
    ELSE (* ignore *)
    END
END PowerHandler;

PROCEDURE AllocPower*;                                    (* allocator for loading *)
    VAR p: Power;
BEGIN
    NEW(p); Expressions.Deposit(p, PowerHandler)
END AllocPower;

PROCEDURE NewPower*(base, exponent: Expressions.Expression): Power;
    VAR p: Power; r: Expressions.Rider;
BEGIN
    ASSERT((base # NIL) & (exponent # NIL));
    Expressions.OpenRider(r, Expressions.emptyList);      (* generate successors list *)
    Expressions.Insert(r, base, 0); Expressions.Insert(r, exponent, 0);
    NEW(p);
    Expressions.Init(p, PowerHandler, Expressions.PowerBind, 0, Expressions.ThisList(r));
    p.base := base; p.exponent := exponent;               (* set fields for direct access *)
    RETURN p
END NewPower;
```

Before loading an object of type *Power* from a file the command *ExprStd.AllocPower* must be called to allocate a new object. (The expression type *Power* is implemented within Module *ExprStd*.) The message handler returns the command's name upon receipt of an *IdentifyMsg*.

Canonical normalization requires no activity, since the representation of an exponentiation is unique. As the only options, the expression could be simplified to its basis if the exponent is known to be 1, or it could be replaced by a 1 if the exponent is known to be 0. We have generally not included such simplifications into canonical normal from computation.

## 5.3 Expression Drawing

Section 3.5 showed how to extend the framework towards expression drawing and editing. The main results were that the program drawing an expression must be bound to that expression object and that bounding boxes are needed to abstract from concrete graphical presentations. Drawing ports are abstractions of graphic devices. They are used to make the individual drawing programs device independent.

### 5.3.1 Drawing Ports

Drawing ports are a commonly used abstraction for graphic devices. Many operating systems feature such an abstraction as a basic service. Oberon does not.

In the expressions framework, the draw ports are needed because of the extensibility requirements. The framework should be independently extensible in different dimensions: any new expression type should be composable with any new device type at run time. Hence, a common interface is needed for all drawing devices.

Even if the set of devices is assumed to be fixed, an abstraction is needed in order to allow for new expression types to be programmed with reasonably small effort. In most practical settings the system will use two devices for output: a screen and a printer. Without an abstraction of these devices, every expression type would need to contain two drawing programs: one for the screen and one for the printer.

This duality can be found elsewhere in the Oberon System. Text Elements, for instance, need to deal with two different messages and access the two devices directly. This has one advantage: the graphical appearance may depend on the device used. On the other hand, both devices are bitmaps.

These differ in their resolution only. The differences are not significant enough to justify a completely different treatment. The port's resolution can be made accessible to the port's user, thus allowing for resolution dependent algorithms.

When trading the possibility of special device treatment against the simplicity of extension with expression types, we have decided in favor of the second. The framework offers the abstraction of drawing ports to be used for drawing.

The next design decision to make is how the abstraction should look like.

A port is an object. It abstracts a bitmap of a particular size with a particular resolution. These parameters must be made available to the port's user.

Secondly, a set of operations to draw on a port must be defined. Defining this set is the critical part. If too few operations are included, using the port is too cumbersome. If too many operations are included, implementing a port requires too much effort.

For the implementation in Oberon the set of operations has been kept very small. It has been chosen by considering the operations implemented by the actual device drivers (i.e. by Modules *Display* and *Printer*). Interestingly, when programming concrete expressions, no need arose to extend the set of primitive operations. More complex operations, like drawing parentheses, have been introduced as library routines at a higher abstraction level (see Section 5.3.2).

Only two kinds of drawing operations have been defined. The graphic primitive *ReplConst* fills a rectangular area with a color, specified as a parameter. This primitive can also be used directly to efficiently draw single dots, and vertical or horizontal lines. The latter are what is mostly needed to display mathematical expressions. For easy drawing of curved objects, like integral signs or parenthesis, splines could be helpful. Currently, these algorithms rely on drawing individual dots.

The second kind of operations draws characters, using Oberon's font module. For the sake of efficiency, an operation to draw an entire string has been included. (*DrawChar* is only needed to cope with the incompatibility of single characters and arrays of characters.)

When calculating the size of graphical objects the size of characters and strings must be known. Access to the actual font data is hidden by the port abstraction. Therefore, ports offer functions to retrieve the width of characters or strings. Height information is considered to be an attribute of the font, independent of the actual characters. Apart from the overall height, the position of the base line must be available. In analogy to Oberon's

Module *Fonts*, font metric information consists of three items: entire height of the font, maximal extension below the base line, and maximal extension above the base line. Such font metric information can be retrieved from the port.

Two heuristic services are also available from ports. The first one tries to find a font of the same family but smaller than a given one. If no such font can be found, the same font is suggested. Such a service is frequently used when typesetting expressions. Parts to be typeset in a font smaller than their environment are for instance, numerator and denominator of a fraction, exponents, indices, bounds of integrals and limits.

The second heuristic service suggests a width for lines to go well with a given font. This line width is used, for instance, to draw the line in a fraction.

The following shows the Types *Ports* and *FontMetric* as they appear in the interface of Module *ExprViews*:

```
IMPORT
    Fonts;

TYPE
    FontMetric = RECORD
        minY, maxY, height: LONGINT
    END;

    Port = POINTER TO PortDesc;
    PortDesc = RECORD
        unit, w, h: LONGINT;
        PROCEDURE (p: Port) ReplConst(x, y, w, h: LONGINT; col: INTEGER);
        PROCEDURE (p: Port) DrawChar
            (ch: CHAR; x0, y0: LONGINT; fnt: Fonts.Font; col: INTEGER);
        PROCEDURE (p: Port) DrawString
            (s: ARRAY OF CHAR; x, y: LONGINT; fnt: Fonts.Font; col: INTEGER);
        PROCEDURE (p: Port) CharWidth(ch: CHAR; fnt: Fonts.Font): LONGINT;
        PROCEDURE (p: Port) StringWidth(s: ARRAY OF CHAR; fnt: Fonts.Font): LONGINT;
        PROCEDURE (p: Port) GetFontMetric(fnt: Fonts.Font; VAR metric: FontMetric);
        PROCEDURE (p: Port) SmallerFont(fnt: Fonts.Font): Fonts.Font;
        PROCEDURE (p: Port) LineWidth(fnt: Fonts.Font): LONGINT
    END;
```

### 5.3.2 Boxes

According to Section 3.5, drawing an expression is done in two steps. First, a tree structure of bounding boxes is build. Secondly, a graphics can be generated according to this data structure.

To reflect this order of operations, the drawing procedures are not bound to the expression objects, but to the box objects, generated by them. To give access to the respective expression's data, a reference to the expression object is included into the box object.

To allow for replacing expressions selected graphically (as demanded by Section 3.6) further information must be stored with each box. The replacement algorithm needs to be able to construct a path from an expression graph's root to the selected node (see also the substitution operation described in Section 3.3). For each node on that path, the entry in the successor's list must be identified, which contains the reference to the next node on the path. To allow for a bottom up construction of this path, two extra information items must be available with each box: the surrounding box (its ancestor in the box tree), and the entry number in the successor's list of the ancestor expression (called *refNo* in the implementation).

A box does not need to represent an entire (sub-) expression. It can also represent a single, graphical object, like a left or right parenthesis, or a line. This simplifies the drawing process, since all the layout is done while generating the box structure. To include parentheses, for instance, two extra boxes are included into the data structure. These will draw the parentheses automatically during the second step. Storing extra information with the expression's box, whether to draw parentheses and where to place them, is not needed.

The module *ExprViews* introduces Type *Box* as an extensible base type. Objects of type *Box* are never changed. However, for the sake of simplicity we resisted the temptation of making boxes immutable like expressions. Two reasons shall be given for this. On one hand, there is no reason to change an existing box object. On the other hand, boxes are derived from expressions on demand. Damaging the box structure would be much less harmful, than damaging the original data.

The following excerpt from the interface of Module *ExprViews* shows the definition of Type *Box*. To have a box structure constructed by an expression object, a message (*GetBoxMsg*) must be sent to it. This is done by the library procedure *ExprBox*. When calling *ExprBox*, an expression, the surrounding box (if any), the reference number in the higher level expression's successor's list, the draw port, and the font to be used must be specified.

Further library procedures offer services, to create special boxes of general importance. These are boxes to display strings (*StringBox*), brackets, parentheses, or curly braces (*BracketBox*), and a symbol for an ellipsis used for elisions as discussed in Section 4.1.3 (*EllipsisBox*).

```
IMPORT
    Fonts, Expressions;

    Box = POINTER TO BoxDesc;
    BoxDesc = RECORD
        next, desc, asc: Box;
        refNo: LONGINT;
        exp: Expression;
        x, y, w, bot, top: LONGINT;
        fnt: Fonts.Font;
        draw: PROCEDURE(box: Box; port: Port; x, y: LONGINT; col: INTEGER)
    END;

    GetBoxMsg = RECORD(Expressions.Message)
        port: Port;
        depth: LONGINT;
        fnt: Fonts.Font;
        box: Box
    END;


PROCEDURE ExprBox(e: Expression;
    asc: Box; refNo: LONGINT;
    p: Port; depth: LONGINT; fnt: Fonts.Font
): Box;

PROCEDURE StringBox(s: ARRAY OF CHAR; asc: Box; p: Port; fnt: Fonts.Font): Box;
PROCEDURE BracketBox(kind: CHAR; desc: Box; p: Port; fnt: Fonts.Font): Box;
PROCEDURE EllipsisBox(asc: Box; p: Port; fnt: Fonts.Font): Box;
```

### 5.3.3 Selections

Section 3.6 described how to extend the framework towards a user interface for computational commands. The programming interface to be used by such commands has to offer two mechanisms: one to retrieve the current selection and one to update the selected expression. For identification of the originally selected expression, a special object is used to represent the selection.

It is a fundamental design goal of this framework to allow for independent extensions with computing commands and with displaying and editing components. An abstract type for selections is the interface between these two groups. Each editor implements objects of this type and each computing command uses one.

#### Using Selections

The active selection can be retrieved through a function, *LatestSelection*. Its result is an object of type *Selection*. If no selection exists, *NIL* is returned.

The selection object refers to the selected expression. It further contains a time stamp, indicating when the expression had been selected.

To replace the selected expression, the computation's result must be assigned to the expression field. Afterwards a notifier procedure must be called, also provided by the selection object. This will cause the update of the editor. At the same time, a new selection object is returned, which may be used for further computation on the same value.

## Implementing Selections

Every editor implements a notification procedure. This is the replace mechanism. The type *Selection* can be extended to keep additional information which is available with the selection and has to be used for replacing. Examples of such information are a reference to the icon holding a selection and the selected bounding box.

Generating selection objects and making them available to the generic selection retrieval mechanism has a problem. If several editors are used at the same time, all of them must be considered when searching for the current selection. If more than one selection is found, the time fields of the selection objects are used to determine the current selection.

It would break independent extensibility if the selection consumers, i.e. the computing commands, have to poll all selection producers, i.e. the editors. Selections must be retrieved from a central service. Editors have to register themselves with that service once.

Usually, such registrations are not necessary at all within the Oberon System. As a rule, editors are implemented as display frames. Selections are retrieved by broadcasting a request message to all visible display frames. The frames, which hold a selection, change the data in the message record accordingly.

This rule of Oberon is violated if text is used to compose parameters for a computation, as discussed in Section 4.1.4. In this case, text selections must be recognized, too. Consequently, the selection retrieval mechanism would need to request text selections as well as expression selections. This would require to plan ahead for using text when designing the general, editor independent selection mechanism. Instead, the text-based expression editor shall be seen as an extension component, which is introduced later.

The part of the interface of Module *ExprViews*, that deals with selections, is given below.

```
IMPORT Expressions;

TYPE
    Selection = POINTER TO SelectionDesc;
    Notifier = PROCEDURE(this: Selection; VAR newSel: Selection);
    SelectionDesc = RECORD
        exp: Expressions.Expression;
        time: LONGINT;
        promise: BOOLEAN;
        notify: Notifier
    END;

    Selector = POINTER TO RECORD
        selTime: PROCEDURE(): LONGINT;
        thisSelection: PROCEDURE(): Selection
    END;


PROCEDURE LatestSelection(): Selection;

PROCEDURE RegisterSelector(s: Selector);
PROCEDURE RemoveSelector(s: Selector);
```

An example how to use selections is the substitution command presented in Section 5.4.

### Implementing Scripting

The selection mechanism is also used to implement the scripting facility discussed in Section 4.2.4. As a result, any command, that can be used interactively, can also be used within a script.

Script execution involves passing the operands to the commands. This is easily done by registering a special selector, that feeds the data into the selection retrieval mechanism.

The flow of control of the script is steered by the notification procedure. It gets control, when a computation is finished and it can issue the call of the next command.

### 5.3.4 Drawing Expressions Of Type Power

The following shows what is to be added to the implementation of Type *Power*, already discussed in Sections 5.1.4 and 5.2.2. The program constructs the box trees for the power's base and exponent. These two are combined within a new box, representing the power expression itself. The base is put into parenthesis if its binding power is lower or equal to that of an exponentiation. For the exponent a smaller font is used.

It is noteworthy, that the box representing the expression itself does not carry any further information. To draw an expression of type *Power* requires simply to draw all its successors.

The procedure's parameter *depth* is used to implement a simple elision mechanism: every expression is only displayed up to a certain depth in structure. An ellipsis is shown instead of a subexpressions if this is nested deeper than that *depth*.

```
PROCEDURE PowerBox(
    pow: Power; port: ExprViews.Port; depth: LONGINT; fnt: Fonts.Font
): ExprViews.Box;
    VAR box, base, exponent: ExprViews.Box;
BEGIN
    IF depth > 0 THEN
        NEW(box);
                                                          (* generate box for base *)
        base := ExprViews.ExprBox(pow.base, box, 0, port, depth - 1, fnt);
        IF pow.base.binding <= Expressions.PowerBind THEN
                                                          (* parentheses around base? *)
            base := ExprViews.BracketBox("(", base, port, fnt)
        END;
        base.x := 0; base.y := 0;
                                                          (* generate box for exponent *)
        exponent :=
            ExprViews.ExprBox(pow.exponent, box, 1, port, depth - 1, port.SmallerFont(fnt));
        exponent.x := base.w;
        IF exponent.top - exponent.bot < base.top THEN    (* vertical offset of exponent *)
            exponent.y := base.top - (exponent.top - exponent.bot) DIV 2 - exponent.bot
        ELSE exponent.y := base.top DIV 2 - exponent.bot
        END;
                                                          (* set up power box *)
        box.desc := base; base.next := exponent;
        box.w := base.w + exponent.w;
        box.bot := base.bot; box.top := exponent.y + exponent.top;
        box.fnt := fnt; box.draw := ExprViews.DrawDesc
    ELSE box := ExprViews.EllipsisBox(NIL, port, fnt)
    END;
    RETURN box
END PowerBox;
```

A single line is to be added to the *WITH* statement of the message handler procedure, to dispatch the *GetBoxMsg* to a call of Procedure *PowerBox*:

```
    | m: ExprViews.GetBoxMsg DO m.box := PowerBox(self, m.port, m.depth, m.fnt)
```

## 5.4 The Substitute Command As A Sample Algorithm

### 5.4.1 The Algorithm

A general command to substitute every occurrence of a certain expression with another one will serve as a complete implementation example for the typical operation discussed in Section 3.3. The substitute command retrieves the current selection and applies the substitution operation recursively to it. To deal with commutative operators the pattern is converted to canonical form. Before comparing the pattern with an actual subexpression, the latter is also converted to canonical form (using the same unification scope). The command can be used, for instance, to manually eliminate common subexpressions or to substitute symbols by numbers.

#### Common subexpression elimination

$\dfrac{2^{1+n}\cdot\left(-\left(s+\sqrt{s^2-4}\right)^{-n-1}+\left(s-\sqrt{-4+s^2}\right)^{-n-1}\right)}{\sqrt{s^2-4}}$ can be simplified by replacing all

instances of $\sqrt{s^2-4}$ with w:

*ExprSubstituter.Substitute* $\sqrt{s^2-4}$ => *w* leads to $\dfrac{2^{1+n}\cdot\left(-(s+w)^{-n-1}+(s-w)^{-n-1}\right)}{w}$.

#### Computing the value of a polynomial

Applying *ExprSubstituter.Substitute x => 3*,to $x^3\cdot y+7\cdot x^2\cdot y^2+\dfrac{2\cdot x\cdot y^3}{5}$

results in $3^3\cdot y+7\cdot 3^2\cdot y^2+\dfrac{2\cdot 3\cdot y^3}{5}$, applying to this the command

*ExprSubstituter.Substitute y =>* $\frac{25}{5}$ leads to $3^3\cdot\left(\dfrac{25}{5}\right)+7\cdot 3^2\cdot\left(\dfrac{25}{5}\right)^2+\dfrac{2\cdot 3\cdot\left(\frac{25}{5}\right)^3}{5}$.

The substitution operation itself is programmed as a recursion on the subexpression graph structure. The recursion terminates if either the entire expression is to be replaced or if it does not have further subexpressions.

The expression to be operated on is passed as a variable parameter. If it is equal to the pattern searched for, the substitute expression is simply assigned to the variable parameter. Otherwise, the same procedure is performed on every subexpression of the current expression. If and only if a subexpression is being changed, the list of subexpressions is modified accordingly. Only in this case a new object is generated to replace the original expression. The new object is a replication of the old one but based on the modified subexpression list.

The structure of the substitute algorithm is characteristic for many operations on expression graphs: it does recursive depth-first traversal, changes the value of a variable-parameter if necessary, and generates new nodes only when necessary. The algorithm produces a new expression graph with a minimal number of new nodes.

## 5.4.2 The Complete Implementation Of A Substitution Command

```
MODULE ExprSubstituter;

IMPORT
    Expressions, ExprViews;

PROCEDURE Subst(VAR exp: Expressions.Expression;
    from, to: Expressions.Expression; s: Expressions.UnificationScope
);
    VAR l: Expressions.List; e: Expressions.Expression; r: Expressions.Rider;
BEGIN
    e := exp; Expressions.Normalize(e, s);
    IF Expressions.Equal(e, from) THEN exp := to          (* substitute exp, if appropriate *)
    ELSE            (* apply recursively *)
        Expressions.OpenRider(r, exp.successors);
        WHILE ~r.eol DO
            e := r.exp; Subst(e, from, to, s);
            IF e # r.exp THEN Expressions.Change(r, e, r.attr) END;
                                                          (* update if necessary *)
            Expressions.Forward(r)
        END;
        l := Expressions.ThisList(r);                     (* build new expression if needed *)
        IF l # exp.successors THEN exp := Expressions.CloneOf(exp, l) END
    END
END Subst;

PROCEDURE ReadPar(VAR from, to: Expressions.Expression);
    ....            (* parse command's parameter text *)
END ReadPar;
```

```
PROCEDURE Substitute*;
    VAR sel: ExprViews.Selection;
        from, to: Expressions.Expression; s: Expressions.UnificationScope;
BEGIN
    ReadPar(from, to);
    sel := ExprViews.LatestSelection();
    IF (to # NIL) & (sel # NIL) & ~sel.promise THEN
        s := Expressions.NewUnificationScope();          (* normal form of pattern *)
        Expressions.Normalize(from, s);
        Subst(sel.exp, from, to, s);                     (* substitute recursively *)
        sel.notify(sel, sel)                             (* update selection *)
    END
END Substitute;

END ExprSubstituter.
```

## 5.5 Textual Expression Presentation

Sections 4.1.4 and 4.3.1 discussed operator composition and editing. These rely on an alternative, textual representation of expressions. The following describes how the framework can be extended to implement conversion to and from text. The basic design has already been outlined in Section 3.5.4.

The main consequence from Section 3.5.4 was not to use an extensible language and an extensible parser. Instead, a closed language is used, which can represent a closed set of standard expression types. Expressions of further types have to be described, using these standard types. After an expression is parsed successfully, registered transformers are activated. The latter can change parts of the expression.

Doing the transformation in two steps separates concerns: text handling and the language parser are independent of the extension mechanism, and vice versa.

### 5.5.1 Standard Types And The Expression Language

Consequently, the implementation consists of two modules. Module *ExprStd* implements all the standard expression types needed to translate an expression stated in the language. Module *ExprLang* imports Module *ExprStd* and implements a recursive descent parser and a writing procedure. The output of the writing procedure must be valid input to the parser.

The parser is able to deal with a specific type of Text Elements, namely expression icons. These are implemented in the separate Module *ExprIcons*. It implements for such icons: expression representation (including concrete draw ports), selection tracking, and copy over of selections.

An expression icon can refer to an expression of any type. Whenever the parser reads such a text element, the reference carried by it is included into the expression graph under construction. Similarly, when writing an expression to text, expressions of unknown type are represented through icons. (All expression types not defined within Module *ExprStd* are considered unknown here.)

The following expression types are defined by Module *ExprStd*: integers (of arbitrary precision), symbols, named functions, collections (to represent bags and sets), and indices. Some standard infix operators are supported also: addition, subtraction, multiplication, division, and exponentiation.

This is reflected by the following excerpt of the interface of Module *ExprStd* defining all the standard expression types and generators for them. For binary and monadic operators more general types are defined. The text parser, however, can generate only sums and products. Special generators are provided for these important cases.

```
IMPORT Integers, Expressions;

CONST PlainOpAttr = 1; InvOpAttr = 2;                  (* attribute definitions *)

TYPE
   Name = ARRAY 48 OF CHAR;                            (* various expression types *)

   Integer = POINTER TO RECORD(Expressions.ExpressionDesc)
      val–: Integers.Integer
   END;

   Symbol = POINTER TO RECORD(Expressions.ExpressionDesc)
      name–: Name
   END;

   MonOp = POINTER TO RECORD(Expressions.ExpressionDesc)
      name–: Name;
      operand–: Expressions.Expression
   END;

   BinaryOp = POINTER TO RECORD(Expressions.ExpressionDesc)
      plainName–, invName–: Name;
      nofTerms–: LONGINT
   END;

   Power = POINTER TO RECORD(Expressions.ExpressionDesc)
      base–, exponent–: Expressions.Expression
   END;

   Function = POINTER TO RECORD(Expressions.ExpressionDesc)
      name–: Name;
      nofPar–: LONGINT
   END;
```

```
    Indexed = POINTER TO RECORD(Expressions.ExpressionDesc)
        expression–: Expressions.Expression;
        indices–: Expressions.List;
        nofIndices–: LONGINT
    END;

    Collection = POINTER TO RECORD(Expressions.ExpressionDesc)
        size–: LONGINT
    END;


    (* generators for standard Types *)

PROCEDURE NewInteger(val: Integers.Integer): Integer;
PROCEDURE NewSymbol(name: ARRAY OF CHAR): Symbol;
PROCEDURE NewMonOp
    (name: ARRAY OF CHAR; binding: SHORTINT; op: Expressions.Expression): MonOp;
PROCEDURE NewBinaryOp
    (plain, inv: ARRAY OF CHAR; binding: SHORTINT; terms: Expressions.List): BinaryOp;
PROCEDURE NewPower(base, exponent: Expressions.Expression): Expressions.Expression;
PROCEDURE NewFunction(name: ARRAY OF CHAR; par: Expressions.List): Function;
PROCEDURE NewIndexed(exp: Expressions.Expression; indices: Expressions.List): Indexed;
PROCEDURE NewCollection(elements: Expressions.List): Collection;


    (* generators for some specific types *)

PROCEDURE NewNegation(op: Expressions.Expression): Expressions.Expression;
PROCEDURE NewSum(terms: Expressions.List): Expressions.Expression;
PROCEDURE NewProduct(factors: Expressions.List): Expressions.Expression;
```

The parser is a simple recursive descent parser for the expression grammar defined below. An entity has to be an *expression*, to be correctly parsable.

```
expression := ["–"] term {addop term}.
term := factor {mulop factor}.
factor := {atom "↑"} atom.
atom := ("(" expression ")" | subexpression | function | collection | symbol | integer) [index].
function := name "(" [sequence] ")".
collection := "{" [sequence] "}".
index := "[" sequence "]".
sequence := expression {"," expression}.
symbol := name.
name := letter {letter | digit}.
integer := digit {digit}.
addop := "+" | "–".
mulop := "*" | "/".
subexpression := icon.
```

### 5.5.2 Extending The Text Converter

To extend the conversion mechanisms to deal with a new expression type, a representation of the latter must be defined using the types from Module *ExprStd*. Most of the time, named functions will be used to represent such new expression types. Integrals, for instance, are represented by a function named *int*. Other possibilities exist, too. Type *Root*, for instance, is translated to and from a power with a fraction as the exponent:

$$x\uparrow(1\ /\ n) \quad <-> \quad x^{\frac{1}{n}} \quad <-> \quad \sqrt[n]{x}$$

Transformer procedures must be registered with Module *ExprStd* to be called after a text has been parsed. Therefore, an appropriate procedure type plus registration and removal procedures are exported.

To retrieve an equivalent representation of an expression, but based on standard types only, an *ExpansionMsg* is sent. The object's message handler must store the expanded representation in that message record. The function procedure *ExpansionOf* retrieves an expanded representation of an expression (by sending an *ExpansionMsg*). Subexpressions must be converted recursively.

```
IMPORT Expressions;

TYPE
   Substitution = PROCEDURE(VAR exp: Expressions.Expression);

   ExpansionMsg = RECORD(Expressions.Message)         (* to retrieve std. representation *)
      exp: Expressions.Expression
   END;

PROCEDURE Register(s: Substitution);                  (* transformer registration *)
PROCEDURE Remove(s: Substitution);

PROCEDURE DoSubstitutions(VAR exp: Expressions.Expression);

PROCEDURE ExpansionOf(e: Expressions.Expression): Expressions.Expression;
```

The following shows the respective part of the message handler procedure for integral types.

```
PROCEDURE IntegralHandler(e: Expressions.Expression; VAR m: Expressions.Message);
    VAR self: Integral;
BEGIN
    self := e(Integral);
    WITH m: Expressions.IdentifyMsg DO
        ...
    | m: ExprStd.ExpansionMsg DO
        m.exp := ExprStd.NewFunction("int", self.successors)
    ELSE (* ignore *)
    END
END IntegralHandler;
```

The following is the example for an installable transformer in the opposite direction. It replaces functions with the name *int* and 2 arguments by an indefinite integral. Functions with the name *int* and 4 arguments are replaced by a definite integral.

```
PROCEDURE InitIntegral(int: Integral);
    VAR n: LONGINT; r: Expressions.Rider;
BEGIN
    n := Expressions.LengthOf(int.successors);
    ASSERT((n = 2) OR (n = 4));
    Expressions.OpenRider(r, int.successors);
    int.integrand := r.exp; Expressions.Forward(r); int.var := r.exp;
    ASSERT(ExprStd.ExpansionOf(int.var) IS ExprStd.Symbol);
    IF n = 4 THEN
        Expressions.Forward(r); int.low := r.exp;
        Expressions.Forward(r); int.high := r.exp
    ELSE e.low := NIL; e.high := NIL
    END
END InitIntegral;

PROCEDURE SubstituteIntegrals(VAR exp: Expressions.Expression);
    VAR n: LONGINT; int: Integral; l: Expressions.List;
BEGIN
    IF (exp IS ExprStd.Function) & (exp(ExprStd.Function).name = "int") THEN
        n := exp(ExprStd.Function).nofPar;
        IF (n = 2) OR (n = 4) THEN
            NEW(int);
            Expressions.Init(int, IntegralHandler, Expressions.ProdBind, 0, exp.successors);
            InitIntegral(int);                                  (* set direct access fields *)
            exp := int
        END
    END
END SubstituteIntegrals;
```

## 5.6 Binomial Coefficients As A Sample Expression Type

The following module implements a complete expression type: binomial coefficients. They are presented as

$$\binom{n}{k}$$

Computing software can access the subexpressions of a *Binomial* directly through read-only exported record fields. Setting a Rider on the list of successors is needed only for generic algorithms.

The concepts have been presented already for the Type *Power* in Section 5.1 to 5.3. Therefore, a detailed discussion is omitted here.

```
MODULE Binomials;

IMPORT Fonts, Expressions, ExprViews, ExprStd;

TYPE
  Binomial* = POINTER TO RECORD(Expressions.ExpressionDesc)
    n-, k-: Expressions.Expression                    (* direct access to subexpressions *)
  END;

PROCEDURE BinomialBox(bin: Binomial;
  port: ExprViews.Port; depth: LONGINT; fnt: Fonts.Font
): ExprViews.Box;
  VAR u, w, d, h: LONGINT; fnt1: Fonts.Font;
    box, n, k: ExprViews.Box; m: ExprViews.FontMetric;
BEGIN u := port.unit;
  IF depth > 0 THEN
    NEW(box); fnt1 := port.SmallerFont(fnt);
    n := ExprViews.ExprBox(bin.n, box, 0, port, depth - 1, fnt1);
    k := ExprViews.ExprBox(bin.k, box, 1, port, depth - 1, fnt1);
                                                      (* horizontal placements *)
    IF n.w > k.w THEN
      w := n.w + port.CharWidth(" ", fnt1) ELSE w := k.w + port.CharWidth(" ", fnt1)
    END;
    n.x := (w - n.w) DIV u DIV 2 * u; k.x := (w - k.w) DIV u DIV 2 * u;
                                                      (* vertical placements *)
    port.GetFontMetric(fnt, m); d := m.maxY DIV 3 DIV port.unit * port.unit;
    h := port.LineWidth(fnt);
    n.y := d + 2 * h - n.bot; k.y := d - 2 * h - k.top;
                                                      (* set up binomial box *)
    box.desc := n; n.next := k;
    box.w := w; box.bot := k.y + k.bot; box.top := n.y + n.top;
    box.fnt := fnt; box.draw := ExprViews.DrawDesc;
    box := ExprViews.BracketBox("(", box, port, fnt)  (* parentheses around binomial *)
  ELSE box := ExprViews.EllipsisBox(NIL, port, fnt)
  END;
  RETURN box
END BinomialBox;
```

```
PROCEDURE InitBinomial(b: Binomial);                    (* set n and k fields *)
    VAR r: Expressions.Rider;
BEGIN
    ASSERT(Expressions.LengthOf(b.successors) = 2);
    Expressions.OpenRider(r, b.successors); b.n := r.exp;
    Expressions.Forward(r); b.k := r.exp
END InitBinomial;

PROCEDURE BinomialHandler(e: Expressions.Expression; VAR m: Expressions.Message);
    VAR self, copy: Binomial;
BEGIN self := e(Binomial);
    WITH m: Expressions.IdentifyMsg DO
        m.mod := "ExprBinomials"; m.proc := "AllocBinomial"
    | m: Expressions.FileMsg DO
        IF ~m.store THEN InitBinomial(self) END
    | m: Expressions.CloneMsg DO
        NEW(copy); Expressions.Init(copy, self.handle, self.binding, 0, m.successors);
        InitBinomial(copy); m.clone := copy
    | m: Expressions.TestMsg DO
        m.equal :=
            (m.with IS Binomial) & Expressions.EqualLists(m.with.successors, self.successors)
    | m: ExprViews.GetBoxMsg DO
        m.box := BinomialBox(self, m.port, m.depth, m.fnt)
    | m: ExprStd.ExpansionMsg DO
        m.exp := ExprStd.NewFunction("binomial", self.successors)
    ELSE (* ignore *)
    END
END BinomialHandler;

PROCEDURE AllocBinomial*;                               (* allocator for load *)
    VAR b: Binomial;
BEGIN NEW(b); Expressions.Deposit(b, BinomialHandler)
END AllocBinomial;

PROCEDURE NewBinomial*(n, k: Expressions.Expression): Binomial;
    VAR b: Binomial; r: Expressions.Rider;
BEGIN
    Expressions.OpenRider(r, Expressions.emptyList);
    Expressions.Insert(r, n, 0); Expressions.Insert(r, k, 0);
    NEW(b); b.n := n; b.k := k;
    Expressions.Init(b, BinomialHandler, Expressions.AtomBind, 0, Expressions.ThisList(r));
    RETURN b
END NewBinomial;
                                                        (* substitute, installed in ExprStd *)
PROCEDURE SubstituteBinomials(VAR exp: Expressions.Expression);
    VAR b: Binomial;
BEGIN
    IF (exp IS ExprStd.Function)
        & (exp(ExprStd.Function).name = "binomial") & (exp(ExprStd.Function).nofPar = 2)
    THEN
        NEW(b);
        Expressions.Init(b, BinomialHandler, Expressions.AtomBind, 0, exp.successors);
        InitBinomial(b); exp := b
    END
END SubstituteBinomials;
```

```
PROCEDURE InstallBinomials*;
BEGIN
    ExprStd.Register(SubstituteBinomials)
END InstallBinomials;

PROCEDURE RemoveBinomials*;
BEGIN
    ExprStd.Remove(SubstituteBinomials)
END RemoveBinomials;

BEGIN InstallBinomials
END Binomials.
```

## 5.7 Communicating With Maple

A communication link to Maple has been implemented as an example for a component, that communicates with a computation server. This presents a proof of concept for integration of concurrency into the editor-based user interface as introduced in Section 4.1.6.

The Maple link component implements a single command to be activated by the user (or by the script interpreter). The command retrieves the current selection through the selection retrieval and replacement mechanism explained in Section 5.3.3. It immediately replaces the selection by an icon to represent the future result and queues the request for transmission to Maple.

The text sent to Maple is more than the selected expression. This gives the user more flexibility when communicating with Maple. The command parses the text following its name according to the Oberon System's conventions. This text is what is sent to Maple. The text may contain icons representing placeholders. (These are the same as used for the overlay implementation described in Section 4.3.3). For these placeholders the selected expression is substituted.

The main task of the link component is the conversion of expressions. A selected expression must be transformed into text for transmission through the communication channel. This text is being interpreted by Maple. Similarly, the result sent by Maple has to be converted to expression objects.

A communication standard like OpenMath [OM–96] would help as a platform for independent extension in the two dimensions of expression types and server programs. OpenMath was not ready for use at the time the prototype was built. Therefore, a very rudimentary approach has been taken. The same converters are used as for the textual user interface. Maple was changed to deal with the framework's language.

Independent extensibility reaches a fundamental limit here. As long as client and server are not both based on the same extensible framework, extensions must be done on both sides. Each expression type introduced into the server must also be introduced into the client (and vice versa). This problem can only be avoided by either fixing the set of expression types or by using the same extension components on both sides (client and server). The latter is not possible when existing computer algebra systems are used as servers.

# 6 Related Work

Many other projects are related with this thesis. The concepts discussed so far embrace two aspects – component software and document-centered software. The first subsection of this section deals with alternative approaches to improve the architecture of mathematical software. The second subsection focuses on user interfaces for mathematical software, in particular on those related to document-centered software.

## 6.1 Alternatives To Component Software

Components separate concerns. They must be composable at run time and must cooperate where required. Their interfaces must be specified using some standard. Safety requirements have an impact on the abstractions to be used. (See Section 2.2.4 for an introduction to component software.)

Currently, most existing software is not structured by components. It is hard to restructure existing software that does not meet the above requirements. As a consequence, algorithms need to be reimplemented.

It is often claimed that the cost to reimplement mathematical algorithms is too high. Therefore, attempts have been made to improve the software structure without requiring algorithms to be reimplemented. These approaches can be divided into two classes. The first class turns a monolithic application program into an extensible system. The second class uses existing application programs as components of even bigger systems.

All these approaches do not lead to independently extensible component software. In general, application programs are not suited to be broken up into components. On the other hand, using them as components includes a lot of unused and redundant functionality into the overall system. Also, extensibility of monolithic programs is restricted: external software cannot be integrated.

As stated in [Wirth–95], in a mature engineering discipline such a situation should lead to redesign of the software rather than to growth of the overall complexity. Abusing interactive application programs as servers, however, may be acceptable and necessary during the migration to component software and during prototyping. The link to Maple implemented for this thesis, is an example of the latter.

Below, two approaches are discussed in detail which improve existing software towards component software.

### 6.1.1 Turning Interactive Application Programs Into Programming Environments

The traditional approach to make an interactive application program extensible is to add programming facilities to it. These are usually centered around a language interpreter. Typically, this interpreter serves for scripting and extension programming at the same time.

Program execution is slower with an interpreted language than with a compiled language. To compensate, critical algorithms are implemented within the interpreter. As a consequence, both the language and the programming environment become very specialized for a particular type of application.

The most prominent examples of mathematical software systems of this kind are Maple [CGGLMW–91], Mathematica [Wolfram–88], and Matlab [Mat–90].

#### MuPAD

A recent version of MuPAD uses the above design principle in an extensible way. [MoOe–95] describes how so-called *Modules* can be added to the kernel. Modules can be loaded on demand and are accessed like the kernel's built-in functionality from MuPAD's interpreted language.

Modules are programmed in the C programming language. They can access any functionality of the underlying operating system and of the MuPAD kernel. The kernel's internal data is unprotected. A macro library is offered to simplify access to the kernel's functionality.

Modules can extend the kernel with fast implementations of mathematical algorithms and with system-related functionality. The kernel itself is not implemented in the form of Modules.

## Discussion

Mathematical interactive application programs are not well suited as general component software environments. On one hand they are too heavy-weight for a general component software shell. The kernels are burdened with too much mathematical functionality which would better be extracted into separate components. On the other hand, essential prerequisites for independent extensibility are missing (e.g. encapsulation and information hiding mechanisms, or an open user interface to integrate components).

As a rule of thumb, to judge whether a software system may be used as a component software environment, one should ask oneself whether it is possible to implement an independently extensible document editor within it. With the systems mentioned above such an editor would be slow. More importantly, independent extensions are not possible because of the lack of the necessary encapsulation and scoping mechanisms.

A programming environment needs more than a language interpreter, e.g. a program editor and a dynamic loader. As the result, the special purpose application programs are extended by extra functionality, which should better be reused from a lower level.

The individual programming environments cannot be used from within other programs. The approach is ego-centric: everybody declares his/her own environment as the base for extensions. Every extension has to be reimplemented for each such environment.

## A Note On Oberon

The implementation described in this thesis extends the Oberon System. At a first glance, Oberon may be mistaken as an environment of the above kind: it can be extended through its own programming language and it can appear as an application program (extending operating systems like Sun OS, Mac OS, Windows, etc.).

Still, there is an important difference between Oberon and the above environments. Oberon is designed as a component software environment. It is not specialized towards any kind of application. The extension mechanism is based on compilation, making execution of any component as efficient as the kernel. Thus, the inner core can be lean and functionality can be added as needed by extension components. Many such components exist already within Oberon. These can be reused and combined with new ones. This is one of the main advantages of integrating mathematical software into an existing component software environment over creating a special environment for mathematical component software.

### 6.1.2 Software Buses

Attempts have been made to connect existing application programs by communication means. These attempts use the application programs as coarse grained components. This concept and the communication software layer in particular are called *software bus*. For mathematical software, the most prominent example is CAS/PI [Kajler–92, Kajler–92a].

When reusing application programs, the data to be manipulated is stored in the workspaces of the individual application programs. Data to be manipulated by more than one program is replicated. Since names are used as references within each workspace, the individual name bindings must be synchronized. This is particularly difficult if servers can be added to the running system at any time. CAS/PI lacks this synchronization.

Using interactive application programs as servers generalizes the concept of pipes in the UNIX operating system, which allows to combine programs by redirecting their input and output streams. This mechanism is suitable for combination of non-interactive programs, which transform sequences of bytes, but it is doubtful whether it is the right tool to accomplish interaction of already existing, self-contained, interactive application programs.

Interactive application programs have not been designed as computation servers but to conduct a dialog with a human user. This makes it hard to analyze the output of a program automatically.

Implementing a connection to Maple for this thesis showed that even small issues create big problems. For instance, it is difficult to identify the end of the output stream or to deal with error conditions. The solution is to modify the interface part of the application program to use a communication protocol instead of the command language. Such protocols for mathematical software are being defined for Mathematica [Wolfram–93] and by the OpenMath group [OM–96].

As an alternative, some application programs are available as linkable libraries with a programming interface. Typically, the library allows the same type of interaction as the interactive application program. To use such a library within a software bus system requires to program stubs that connect the library to the communication-based software bus.

Serializing the data, sending it over a sequential channel, and replicating it in another program's workspace is much less efficient than sharing memory. It requires both time to transform the data and space to replicate it.

Finally, the stream interface is too restricted. It does not allow for static type checking. At best, dynamic type checking can be supported.

Component software should use a common data repository, accessed directly by all components. For the design presented in this thesis, workspace management was separated from computation software. Workspaces can be shared, no serialization or replication of data is necessary, and no synchronization of workspaces is needed.

Remote machines may be used as computation servers, but the servers should be designed with this purpose in mind. (The client/server architecture can be seen as a reduction of the software bus concept.) All state should be kept at the client's site. The servers should be stateless like with CaminoReal [ABMW–88]. (Still, caching may be used to improve the performance.)

The same holds for programming: only when programming is centered at the client's site, it becomes possible to bundle the computational power of different servers in one script or program. Furthermore, the user needs to master only one programming language instead of one per server.

### 6.1.3 Shared Object Libraries

Current operating systems offer a unit of extension that is smaller than programs. Examples are Shared Object Libraries in the Sun OS or Dynamical Link Libraries (DLL) in Windows. They offer dynamic loading and are a prerequisite for component software, but they are not a full alternative. They support neither interface checking, nor shared memory management, nor a foundation to integrate user interfaces. (See 2.2.4 for a discussion of the requirements for component software.)

## 6.2 User Interfaces And Document Editors

The idea to combine document editing and user interfaces is not new. Early and general examples are the Macintosh *Finder* program [InsMac–91] or hypertext systems. Modern examples are graphical user interface builders like Delphi, Visual Basic [Schneider–95], the Gadgets system of Oberon System 3 [Gutknecht–94], or the form subsystem of Oberon/F [OF–94].

In the realm of mathematical software, Maple's *Worksheets* [CGGLMW–91] and Mathematica's *Notebooks* [Soiffer–95] are user interfaces which are based on documents. Examples of document editors that incorporate a link to a symbolic computation system are CaminoReal [ABMW–88] and the Euromath system [Sydow–92].

A complete and comprehensive overview of mathematical user interfaces can be found in [Soiffer–91] and in [KaSo–94]. Therefore, only projects closely related to this thesis will be mentioned in the following.

### 6.2.1 Expressions Within Documents

When turning documents into interfaces to mathematical software, expressions must be incorporated into documents. This has value in itself since it can be used for document processing.

Expression editing can be built into a closed document editor. An example are the extensions of the editor Lara described in [Schär–91]: a fixed set of templates for various mathematical expressions is made available. To introduce a new expression type, configuration files and the program itself have to be changed. The latter involves recompilation. A globally unique number has to be assigned to the new type. This inhibits independent extensibility.

An alternative to modifying a closed document editor is to extend an extensible editor like OPUS [Vetterli–91], Framemaker [Framemaker–89], or the editor Tioga of the Cedar System. CaminoReal [ABMW–88] is an extension of Tioga, which also can be connected to Maple (see Section 6.2.2).

With the upcoming industry standards for compound documents (like OpenDoc and OLE) more and more expression editors are made available as document parts. The most prominent example is MathType [MT–87], which is delivered together with Microsoft Word [MW–93]. The integration is based on OLE.

Such document parts aim at editing and document processing. To implement workspaces as suggested by this thesis, a programming interface to containers and parts is needed. No user interface software has yet been based on such building blocks.

Much document preparation is currently done by batch-oriented typesetting software, like TeX [Knuth–84] or LaTeX [Lamport–94]. Documents are described in a special language. This description is translated to a two-dimensional representation by a non-interactive program. Batch-oriented systems cannot be used as a base for user interfaces, because they are not interactive.

The advantage of batch-oriented typesetters is the excellent typographical quality of their output. The optimal solution may be to use a batch-oriented typesetting system as an intelligent printer driver. In such a setting the interactive compound document editor is used for editing and as a user interface. For printing, a document description suitable for a batch-oriented typesetter would be generated. Of course, the WYSIWYG property would be lost, but compared to describing a document using TeX or LaTeX the feedback is still better.

### 6.2.2 User Interfaces of Computer Algebra Systems

Traditional user interfaces to mathematical software are based on sequential dialogs, command lines, and text. They can be used through primitive, character-based terminals. Such user interfaces are no longer state of the art. This section lists some more modern examples.

#### Mathematica, Maple

The benefit of putting together document processing and mathematical software has been recognized by the most important computer algebra system vendors. Maple [CGGLMW–91] for instance, uses *Worksheets*. These are documents within a proprietary document processing environment. They can contain mathematical expressions (apart from text and plots), which can be parsed to be input into Maple's name-based workspace. Mathematica's *Notebooks* [Soiffer–95] implement a conceptually similar idea. As an early example of such specialized, interactive document editors also Mathcad [MC–93] should be mentioned.

These examples show that it is appealing to use documents as user interfaces. In the case of Maple and Mathematica, document processing was added to the user interface of symbolic computation programs. These programs cannot be combined with each other nor with other document processing software.

The project underlying this thesis takes the opposite approach. The user interface is built on an existing document processing environment. Documents are used as the universal integration platform for a variety of user interfaces. Furthermore, the user interface for mathematical software is not bound to an individual computation program. On the contrary: the user interface becomes the master who delegates to computation slaves. Several of such slaves can be used from a single interface concurrently.

Mathematica's *Notebooks* and Maple's *Worksheets* are only front ends for input and output. They are not the data repositories accessed directly by the computation engines. Both Mathematica and Maple use hidden state (e.g. name bindings) to implement workspaces. The documents within the user interface are only used as a special kind of terminal.

This thesis suggests to use documents as workspaces directly. The hidden name bindings are avoided entirely. Much of the functionality needed for workspaces can be reused from the document processing software, leading to a lean system.

### CaminoReal, Euromath

The projects most similar to the one underlying this thesis are CaminoReal [ABMW–88] and the Euromath system [Sydow–92]. Both extend a general document editor with mathematical expressions and both allow access to computation software.

CaminoReal is based on the extensible editor Tioga of the Cedar system. An abstract syntax is used to represent expressions internally. A local library as well as remote servers can be used for computation. The problem of representing concurrent computation within an interactive document editor is stated but not resolved. Evaluation of a contiguous document section is suggested for scripting, but no local scopes are created. On the contrary, a global name binding table reintroduces all the problems of hidden state and the dependences on session history.

For the Euromath project an entire extensible structure editor has been developed. Expressions are a special type of document structure. The client/server scheme is suggested to integrate existing application programs as computing servers. This raises many of the problems discussed for software buses in Section 6.1.2. Neither composed parameters, nor concurrent computations, nor scripting are addressed.

### Mathcad

For the user interface of Mathcad [MC–93] a special document editor has been implemented. Documents are composed from text, pictures, express-ions, plots, etc. Computation commands can be applied to expressions. Expressions and plots can be related as with cells of a spread sheet. Mathcad features its own library of numerical algorithms and a link to Maple for symbolic computation.

### Theorist

Theorist [BoWa–89] also uses a document model as front end to its computation engine. Computing with Theorist means to edit a document. The latter presents one or several *Theories*. Each *Theory* represents one branch in a tree of manipulations. Whenever case-analysis leads to more than one possible result, The tree branches.

The user interface of Theorist is a constrained direct manipulation editor for equations. Equations can be manipulated directly. It is possible, for instance, to select a term within a sum on one side of an equal sign and to drag it to the other side of the equal sign. This is interpreted as the command to subtract that term on both sides of the equation.

This interface is very intuitive for simple operations. More complicated operations are available through menu commands.

With workspaces implemented through compound documents such a direct manipulation constrained editor could be wrapped into a document part.

### 6.2.3 Plotting And Help Texts

[KaSo–94] states that user interfaces for symbolic computation software must feature more than expressions. In particular, interfaces to display plots of mathematical objects and the integration of help texts is demanded.

This thesis treats these issues as separated concerns. A software component generating plots will have to use a special document part to display its plots. Such document parts can be inserted into the same documents as expressions can. Apart from this, plotting is orthogonal to the material discussed in this thesis.

The same holds for help texts. These can be provided as documents, independent of the rest of the software. Existing hypertext facilities can be used to organize them. A help text may contain parts to represent expressions. This is advantageous, as the user can work with examples within the help document or extract data from it. Again: the help system is integrated into the document processor instead of rebuilding a document processor as help system.

### 6.2.4 General Object-Oriented User Interfaces

There exists a tremendous number of tools for the interactive design of object-oriented user interfaces. Just a few such environments are [LVC–89, Calder–90, Gutknecht–94, OF–94, Schneider–95]. All these tools have a different target than this thesis. They allow to combine small editors for single objects (widgets, controls, etc.) on a dialog panel. In this thesis the focus is different: documents are used as (persistent) object stores allowing direct manipulation.

### 6.2.5 Expression Editors

Part of the framework described in this thesis is an extensible expression editor. It was not the goal to introduce a new editing model for users. The text-based editing model resulted from the overall structure of the framework.

Soiffer presents a guideline for expression editor implementation in [Soiffer–91]. He further offers a very complete review of such editors and describes the editor MathScribe. According to [Soiffer–91] two different mechanisms for entering and editing expressions are to be distinguished: incremental parsing and patterns (templates, overlays).

With incremental parsing, formulae can be entered as text. This text is analyzed after each key stroke and a graphical representation of the expression currently being entered is reformatted if needed.

The second entering mechanism relies on patterns which are applied as templates or overlays. Such patterns can either be selected from a graphical palette, or they can be bound to keys.

Most existing expression editors are based on these concepts, or on a combination of them. Section 4.3 shows how these basic technologies were incorporated into the framework of this thesis.

#### CAS/PI

Kajler already observed in [Kajler–92] that a user interface must be extensible with mathematical objects, in particular if it is to be independent of computation programs. Whenever a new type of expressions is introduced into a computation program or by a new program, it must also be introduced into the user interface.

CAS/PI is an implementation allowing for such extensions. To introduce a new object into CAS/PI its various aspects must be described using several ad-hoc languages. This makes extending CAS/PI a bit cumbersome: many changes at many unrelated locations are needed. The various configuration files of the user interface and the computation software all have to be changed in parallel and consistently.

#### Kaava

For Kaava [Rimey–92] template-based formula entering was studied in detail. Kaava does not support any (incremental) parsing. Instead, templates can be bound to keys on the keyboard. Such templates may contain placeholders in which cases they are used as overlays.

Kaava's editor goes much further than the one implemented for this thesis. It supports insertion points between two expressions. These are automatically extended to selections according to a complicated algorithm. Since Kaava's editing technology is built upon expression trees and bounding boxes, it could be implemented to be used within expression icons.

# 7 Conclusions

Several conclusions can be drawn from the design, implementation, and description of a framework for document-centered mathematical component software. Some of them are related to mathematical software, some of them are related to component software and document-centered software.

## 7.1 Mathematical Software Systems Can Be Lean

The overall conclusions from the presented work is that document-centered mathematical component software is feasible and that it can be lean software yet providing significant functionality. Design and implementation of a prototype have been presented. The implementation in Oberon serves as a proof of concept. It demonstrates that a small core framework is sufficient. It has been successfully used to accomplish small computation tasks during the author's every day's life and for document preparation. This text is an example of the latter.

The implementation includes the abstract and generic expression data type, the drawing framework, integration into text documents, a scripting facility, a generic editor, a link to Maple, some simple computation commands, and a variety of expression types. All this has been implemented with 6720 program statements (see Table 2 in Chapter 5).

Most of the implementation's simplicity is due to the regular and generic design. One might suspect that using small, simple, generic interfaces may cause a significant loss of efficiency. Experience with the implementation shows that this is not the case. No significant delays can be observed by the user. The main reason behind this is probably that only one address space is used, avoiding unnecessary data replication and transmission through byte-oriented channels.

Leanness not only shows in small program size and moderate hardware requirements. More importantly, lean software is easier to understand than fat software. Such understanding is essential for the software scientist and teacher: the scientist needs to extract principles for generalization and the teacher needs to present the essentials to the students. This thesis proves that the framework for document-centered mathematical component software can be described comprehensively.

It may be argued that the prototype is lean, only because it is over-simplified. The response to this is two-fold. On one hand, the prototype is functional, i.e. it contains the features *needed*. There will always be additional features *wanted*. Obviously, there is a trade off between software leanness and user comfort. Finding a reasonable compromise is one of the largest difficulties in software design. On the other hand, addition of some single features to the prototype should not be too costly, i.e. it should not add an order of magnitude to the prototype's size. The claim that the software is lean should hold, even after several additions.

The main tradeoff is between genericity and special treatment of particular expression types. Genericity is the key to leanness. Special treatment of properties is demanded in particular for a few special operands like the standard arithmetic operators. Generalizing these special demands complicates the generic abstractions. It is doubtful whether this price should be paid. It can be concluded that mathematical software is indeed hard to treat generically as long as no concessions are made.

## 7.2 Computation Servers Should Be Stateless

The client/server architecture allows a client to use several computation servers. The client implements the user interface and manages the workspace. Programs that are used as servers should not have additional, separate workspaces.

Before going into detail, it should be noted that these considerations do not apply to systems that are created entirely on the basis of distributed objects. Such systems are equivalent to conventional systems with single address spaces. The considerations do apply to systems with several separated address spaces. Such systems result from reusing existing computer algebra systems or when computing services are offered to a wider range of clients, for instance via the World-Wide-Web.

If computing servers do not share objects, they should be stateless. All the state should be handled by the client site that implements the user interface. The client has to compensate for the loss of functionality caused by making the servers stateless. This lost functionality includes the possibility to bind properties to objects or to use name bindings. Document-centered software should offer alternatives to these. These mechanisms need to be extended to remote computing.

For efficiency reasons, internal state of a server may still be used for caching to reduce communication needs or to keep libraries loaded. It would be the system's (i.e. client's) task to detect inconsistencies and to invalidate the caches as needed. To the user, however, a single workspace must be presented.

The main advantage of stateless servers is that they allow the user to combine computation servers freely. With servers keeping separate state, the user would have to keep the workspaces of the different servers consistent or to cope with inconsistent results.

The same argument holds for programming. Combining operations of several servers within one program becomes cumbersome if the individual server's programming facilities are used. For instance, all communication between servers must be programmed explicitly. Furthermore, if different programs are used as servers, different programming languages must be used and understood.

If programs are executed at the client's site, the user has to manage one programming environment only. Single operations can be delegated to stateless servers, for instance by remote procedure calls. The services themselves should be programmed on the server's site directly, i.e. without involving the client program.

## 7.3 Extensibility Is Better Than Reuse

The most important guideline during the framework's design was to strive for independent extensibility. This contrasts traditional object-oriented programming, which focuses on reusing existing objects.

Traditional object-oriented frameworks implement partial solutions (i.e. functionality). They are specialized later to solve a particular problem. Doing so is seen as reusing the functionality implemented in the framework.

Independent extensible component frameworks are different. They materialize the design rules for components extending it. An example for such a rule is that expression graphs must not be changed. This rule is materialized by an abstract data type for immutable expression graphs. The system's functionality is implemented within the components that extend the framework, not in the framework itself.

In general, extensibility is more important than reusability. With an extensible system the user combines components on demand. These components need not to use each other directly, but they use common abstractions. They can be composed at run time, and still cooperate. Expression graphs, for instance, can contain expression types coming from several components, though these components are not aware of each other.

Extensibility allows for reuse on a higher level, i.e. on the user's level. There is no static relation between components that extend a framework in parallel. The developers of the individual components do not need to know about each other. Only the user decides which of the existing components to use to accomplish a specific task.

## 7.4 Workspaces Can Be Implemented By Compound Documents

Compound documents can be workspaces for users. The workspace is implemented by the document container. The objects within the workspace are represented by document parts. The document editor allows direct manipulation of workspaces. Objects can be inserted and removed from the workspace by inserting and removing the corresponding document parts.

The documents *are* the object repository – in contrast to existing front ends that use documents only as sophisticated terminals to manipulate traditional workspaces based on hidden state.

The traditional way to implement workspaces is to use name bindings. Using compound document parts instead of names has several advantages:

- it can be seen directly, which objects are in the workspace.
- no internal table of name bindings must be memorized by the user. (The extra information possibly provided through a name can be represented in other ways within a document.)
- document parts can carry more information than names, e.g. they can typeset formulae.
- document parts can react on user input, allowing e.g. for selection of subexpressions.
- document parts can represent upcoming results computed by a server.

Name bindings are still needed for scripting. In contrast to workspace implementations, these name bindings must not be global. Instead, each script should generate its own scope.

Computational component software is centered around a repository for objects. Doing so, separates data management and user interface issues from computational components. Such a repository has a very light weight implementation using a compound document framework. The document editor allows for direct manipulation of the workspace. Computing operations are just particularly powerful editing operations.

Compound documents allow for better integration with various software components. On one hand, arbitrary computational programs can share a workspace. On the other hand, arbitrary objects can be bundled in one workspace. This allows for commenting, inclusion of plots, etc. Finally, other document processing software (e.g. for layouting or hypertext linking) can be used with such workspaces. This separates the concerns of plotting and online help among others. The latter, for instance, can be presented as a hypertext document including expressions. A special online help system is no longer required.

## 7.5 Document Parts Should Be Seen In Their Context

The concept of compound documents should be enhanced beyond separation of containers and parts. Parts can be interpreted in the context provided by their current container. Within a text container they can be used as tokens of a formal language.

This has three major applications in the presented framework:

- ad hoc composition of multiple parameters for commands as text.
- scripts as an interpretable document section.
- a simple text-based expression editor can be created with almost no extra implementation effort.

Operands of commands can be stated in one of several forms. The framework presented deals with expressions represented as text, as mixture of text and document parts, as single document parts, or as subexpressions selected within document parts.

## 7.6 Experience With Oberon

The framework has been implemented as an extension of the Oberon System using the Oberon Programming Language. Technically, this environment proved to be very adequate. In the following, some experience is discussed with both the Oberon Language and the Oberon System. Part of the discussion is on specific details. For these, the reader is assumed to be familiar with Oberon.

### 7.6.1 Experience With The Programming Language

The Oberon Programming Language served well to express the abstractions needed for the component framework. Encapsulation of components was provided through modules. The type-checked interfaces allowed to guarantee crucial invariants.

Strong static typing was an important practical aid when implementing the framework. The type-checking compiler was of big help during several major redesigns that involved changes in the interfaces, even of low level components. The compiler could be used to statically identify all the program parts needing adaptation. Such changes become necessary during the design and prototyping of any component framework.

The language Oberon exists in two versions, Oberon and Oberon-2, the latter being a superset of the former. The most important additions introduced by Oberon-2 are three concepts: type-bound procedures, read-only export, and dynamic arrays.

In the mathematical component framework type-bound procedures have been used very rarely. They could easily have been avoided entirely. Instead, the message record handler approach of the original Oberon system has been followed. In some cases (like the draw method for bounding boxes) a procedure variable was used to bind a method. The reason for not using type-bound procedures was that binding methods to objects is often not paralleled by type extension. Type-bound procedures often require a type extension only to allow overriding of a method.

Read-only export has been used frequently to protect abstractions. It is an important tool if variables are used in interfaces. The only safe alternative is not to export the variables (or record fields) at all, but to offer access procedures instead. The latter are less efficient, but they give more freedom when implementing an interface.

Dynamic arrays were rarely used. Only the library module implementing arbitrary precision integer arithmetic relies entirely on dynamic arrays. In some cases, usage of dynamic arrays was not possible because of the implementation restriction that disallows dynamic arrays to hold pointers.

### 7.6.2 Experience With The System Library

The Oberon System was a major source of inspiration and a useful environment for the framework. In particular, the uniform address space made the integration of components (new ones and existing ones) very easy. Memory protection on object level through the programming language proved to be a reasonable substitute for separate address spaces. In its light-weightness this model is preferable to that of OpenDoc, for instance. OpenDoc uses a separate address space for every document. Such border lines hinder reuse of computational components which need not to be assigned to particular documents.

For the expression editor the lightweightness of viewers is essential. In Oberon viewers are simple objects, avoiding the overhead of creating a new process or even of loading and starting a new application. This has direct impact on the performance. Viewers can be opened and closed fast enough to use a separate viewer when editing an expression. No recognizable delays are imposed on the user.

#### Using Text Elements

Text Elements were used extensively and in a way not entirely foreseen by their designer. It turned out that the subfocus mechanism is difficult to understand entirely. To some extent this is also true from the user's perspective. A Text Element's frame can hold a subfocus within its container, though the latter is not focused. Such a Text Element's frame looks like being the current input focus, but it is not.

Furthermore, the concerns of input focus and mouse command interpretation are not well separated. It is necessary to switch between different mouse command interpreters since it must be distinguished between mouse commands to the text editor and such to the Text Element. In Oberon, that switch is tied to the input focus. Therefore, only one Text Element per viewer can interpret special commands. That Text Element is the input focus at the same time. This causes some problems with copy-over of subexpressions within the same document.

To use the existing text model as a base for workspaces, i.e. as data repository, one more facility is missing. When a text is to be stored the DAGs referred to by document parts must be stored with it. In general, these DAGs will be not completely disjoint. Several document parts will refer to nodes in the same DAG. These relationships should be taken into account when storing the document. Otherwise a high extra cost in memory would result, and it would be rather expensive to reconstruct the original graphs when internalizing the document again.

To store such data graphs as a whole, the storing operation must be preceded by an extra step. During this step the entire document can be analyzed. The result of this analysis can be used to store a single graph. References to nodes within that graph are stored with the individual document parts. This requires that the store message sent to the Text Element carries some context. With the current Oberon System, neither preprocessing nor that context are supported. The extension hooks needed are missing.

### Draw Ports Should Be Provided By The System

It has turned out that the Model-View-Controller Separation (MVC) reaches its limits as soon as the Model becomes extensible. This observation can already be made with Text Elements, which allow to extend the text model. Expressions are another example. In all such cases the drawing programs have to be provided together with the model extensions. They cannot be factored out into a separate view component.

Drawing programs should not depend on the graphic devices to be used. If they do, the system is no longer extensible in the dimension of such devices; at least, such extensions are no longer independent from the model extensions. For Text Elements the problem has been circumvented by fixing the set of devices. It may be argued that this restriction may be acceptable because of typical hardware settings. However, the need to program drawing twice can already be cumbersome if drawing is difficult, like for mathematical expressions.

The solution is to provide an abstraction of drawing devices, typically called draw ports. Such an abstraction should be part of the system's core. Present day, extension packages (like Callas [Pfister–93] and Kepler [Templ–94]) define their own draw ports. The expressions framework adds a further port type. This suggests the conclusion that ports are indeed important enough to be generalized for the entire system. This would also simplify cooperation between different packages. Expressions, for instance, could be used within Kepler, too.

## 7.7 Generalizations

Some concepts developed in this thesis have been proven useful beyond mathematical software. For instance, a document-centered user interface client has been implemented for text searching. It communicates with a UNIX-based text searching engine operating on the Oxford English Dictionary. The answer to a query is a set of matches which – in case of a very general query – can have millions of elements. It is necessary to represent such sets as objects to the user. These objects will be used to state query refinements. With similar ideas a user interface to a music data analysis system has been implemented [Eberhardt–91].

Independently extensible systems are a current research topic. This thesis is a case-study in creating independently extensible component frameworks.

## 7.8 Summary

Apart from showing that lean document-centered mathematical component software can be done, the thesis offers the following contributions, which to the best of my knowledge are new and original in this form and combination:

- the concrete design of a component framework for mathematical component software (Chapter 3)
- the suggestion to use compound documents in an integral way, i.e to view parts in the context provided by their container (Chapter 4)
- a document-centered user model for mathematical component software (Chapter 4) including:
  - documents as data repositories, not just as mirrors of internal state kept separately (Section 4.1.2)
  - scripting within documents (Section 4.2)
  - document parts to represent upcoming results of concurrent computing operations within interactive editors (Section 4.1.6)
- a design pattern for immutable, generic graphs (Section 5.1).

# Bibliography

[ABMW–88] D. Arnon, R. Beach, K. McIsaac, C. Waldspurger,
*CaminoReal: An Interactive Mathematical Notebook*,
Proceedings of EP'88, International Conferences on Electronic Publishing, Document Manipulation, and Typography, 1988.

[Ada–83]
*The programming language Ada reference manual*,
The American National Standards Institute (ANSI), LNCS 155, Springer, 1983.

[BAM–94] H. J. Boehm, R. Atkinson, M. Plass,
*Ropes Are Better Than Strings*,
Technical Report CSL–94–10, Xerox Corporation, Palo Alto Research Center, 1994.

[Beech–96] R. Beech,
*The Business Case for Component Software*,
Apple Directions February 96, pp. 19–23, 1996.

[BoWa–89] A. Bonadio, E. Warren,
*Theorist Reference Manual*,
Prescience Corp., 939 Howard St., San Francisco, CA, 1989.

[Brockschmidt–94] K. Brockschmidt,
*Inside OLE 2*,
Microsoft Press, Redmond, Washington, ISBN 1–55615–618–9, 1994.

[Calder–90] P. R. Calder, M. A. Linton,
*Glyphs: Flyweight Objects for User Interfaces*,
Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), 1990.

[CGGLMW–91] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, and S. Watt,
*Maple V Language Reference Manual*,
Springer-Verlag, New York, ISBN 3–540–97622–1, 1991.

[Crelier–94] R. Crelier,
*Separate Compilation and Module Extension*,
PhD Thesis, Diss. No. 10650, ETH Zürich, 1994.

[Deutsch–89] L. P. Deutsch,
*Design Reuse and Frameworks in the Smalltalk-80 System*,
Software Reusability, Vol. 2: Applications and Experience, pp. 57–72. ACM Press Frontier Series, Addison-Wesely, Reading, MA. ISBN 0–201–50018–3, 1989.

[Eberhardt–91] R. Eberhardt,
*MALT: a music searching system*,
Diploma Thesis, ETH Zürich, 1991.

[Fawcett–89] H. Fawcett,
*PAT 3.3 Users.Guide*,
University of Waterloo, Centre for the New Oxford English Dictionary, 1989.

[Framemaker–89]
*FrameMaker Reference Manual, Version 2.0*,
Frame Technology Corporation, San Jose, California, 1989.

[GHJV–95] E. Gamma, R. Helm, R. Johnson, J. Vlissides,
*Design Patterns*,
Addison Wesley, Reading, MA, ISBN 0–201–63361–2, 1995.

[GiRo–74] L. Gilman, A. J. Rose,
*APL – An Interactive Approach*,
Wiley & Sons, New York, 1974.

[Gutknecht–94] J. Gutknecht,
*Oberon System 3: Vision of a Future Software Technology*,
Software – Concepts and Tools, 15:1, 1994.

[InsMac–91]
*The Finder Interface*,
Inside Macintosh, Vol. VI, Apple Computer Inc. 1991.

[JeSu–88] R. D. Jenks, R. S. Sutor
*Axiom: The Scientific Computing System*,
Springer-Verlag, New York, ISBN 0–387–97855–0, 1992.

[Johnson–88] R. Johnson, B. Foote,
*Designing Reusable Classes*,
Journal of Object-Oriented Programming, Vol. 1, No. 2, June, 1988.

[Kajler–92] N. Kajler,
*CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems*,
Proceedings of ISSAC'92, Berkeley, USA, July 1992, ACM Press, 1992.

[Kajler–92a] N. Kajler,
*Building a Computer Algebra Environment by Composition of Collaborative Tools*,
Proceedings of DISCO'92, LNCS 721, Springer-Verlag, Bath, GB, 1992.

[KaSo–94] N. Kajler, N. Soiffer,
*Some Human Interaction Issues in Computer Algebra*,
CAN Newsletter, Vol. 12, March 1994, CAN Expertise Center, Kruislaan 419, 1098 VA Amsterdam, NL, 1994.

[KLMM–94] J. L. Knudsen, M. Lofgren, O. L. Madsen, B. Magnusson,
*Object-oriented environments: The Mjolner approach*,
Prentice Hall, The Object-Oriented Series, ISBN 0–13–009291–6, 1994.

[Knuth–84] D. E. Knuth,
*The TEXbook*,
Addison-Wesley, 1984.
[Knuth–84a] D. E. Knuth,
*Literate Programming*,
The Computer Journal, 27:2, pp. 97–111, 1984.
[KrPo–88] G. E. Krasner, S. T. Pope,
*A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80*
Journal of Object-Oriented Programming, Vol. 1, No. 3, August 1988, pp. 26–49, 1988.
[Lamport–94] L. Lamport,
*LaTeX: A Document Preparation System, User's Guide and Reference Manual, 2ed.*,
Addison-Wesley, Reading, MA, USA, ISBN 0–201–52983–1, 1994.
[LiSh–88] B. Liskov, L. Shrira,
*Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*,
Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia. 1988, ACM SIGPLAN Notices 23 (7), 1988.
[LVC–89] M. A. Linton, J. M. Vlissides, P. Calder,
*Composing User Interfaces with InterViews*,
IEEE Computer, February 1989.
[Magnusson–91] B. Magnusson,
*Code Reuse Considered Harmful*,
Journal of Object-Oriented Programming, Vol. 4, No. 3, November 1991, p. 8, 1991.
[Mat–90]
*Pro-Matlab for Sun Workstations*,
The MathWorks, Inc., 21 Eliot Street, South Natick, MA 01760, 1990.
[MC–93]
*Mathcad 4.0 User's Guide*,
MathSoft Inc., 201 Broadway, Cambridge MA, USA, 1993.
[MoOe–95] K. Morisse, G. Oevel,
*New Developments in MuPAD*,
Computer Algebra in Science and Engineering World Scientific, Singapore, 1995, pp. 44–56, 1995.
[Mössenböck–91] H. Mössenböck,
*The Programming Language Oberon-2*,
Structured Programming, 12:4, 1991.
[MT–87]
*MathType User Manual*,
Design Science, Inc., 4028 Broadway, Long Beach, CA 90803, 1987.
[MW–93]
*Word User's Guide, version 6.0*,
Microsoft Corporation, Microsoft Press, 1993.
[OD–94]
*The OpenDoc White Paper*,
http://www.cilab.org/aboutod.html.

[OED]
*Oxford English Dictionary, 2.ed.*,
Oxford University Press.

[OF–94]
*The Oberon/F User's Guide*,
Oberon microsystems, Inc., Basel, CH, (http://www.oberon.ch/customers/omi), 1994.

[OM–96]
*OpenMath Specification*,
http://www.can.nl/~abbott/OpenMath/, 1996.

[Pfister–93] C. Pfister,
*CALLAS: A Physical Design Framework for Configurable Array Logic*,
PhD Thesis, Diss. No. 9940, ETH Zürich, 1993.

[Reiser–91] M. Reiser,
*The Oberon System – User's Guide and Programmer's Manual*,
Addison-Wesley, Reading, MA. ISBN 0–201–54422–9, 1991.

[Richardson–68] D. Richardson,
*Some Undecidable Problems Involving Elementary Functions of a Real Variable*,
The Journal of Symbolic Logic 33 (4), pp. 514–520, 1968.

[Richardson–69] D. Richardson,
*Solution of the Identity Problem for Integral Exponential Functions*,
Zeitschrift für mathematische Logik und Grundlagen der Mathematik, 15,
pp. 333–340, 1969.

[Rimey–92] K. Rimey,
*Template-based Formula Editing in Kaava*,
Proceedings of DISCO'92, LNCS 721, Springer-Verlag, Bath, GB 1992.

[Schär–91] H. Schär,
*Integrierte interaktive Bearbeitung mathematischer Formeln im Dokumenteneditor Lara*,
PhD Thesis, Diss. No. 9402, ETH Zürich, 1991.

[Schneider–95] D. L. Schneider,
*An introduction to programming using Visual Basic*,
Prentice Hall, ISBN 0–13–191263–1 1995.

[Shneiderman–83] B. Shneiderman,
*Direct Manipulation: A Step Beyond Programming Languages*,
IEEE Computer, Vol. 16, No. 8, pp. 57–69, 1983.

[Soiffer–91] N. Soiffer,
*The Design of a User Interface for Computer Algebra Systems*,
Doctoral Dissertation Report No. UCB/CSD 91/626, Computer Science Division,
University of California at Berkeley, April 1991.

[Soiffer–95] N. Soiffer,
*Mathematical Typesetting in Mathematica*,
Proceedings of ISSAC'95, Montreal, Canada, pp. 140–149, ACM, 1995.

[Stoutmyer–84] D. R. Stoutmyer,
*Which polynomial representation is the best?*
Proceedings of the 1984 MACSYMA User's Conference (1984), V.E. Golden, Ed.,
pp. 221–243, 1984.

[Sydow–92] B. von Sydow,
*The design of the Euromath system*,
Euromath Bulletin 1, 1992.
[Szyperski–92] C. A. Szyperski,
*Write-ing Applications: Designing an Extensible Text-Editor as an Application Framework*,
Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems, TOOLS EUROPE '92, Dortmund, Germany Prentice-Hall, Englewood Cliffs, NJ, 1992.
[Szyperski–92a] C. A. Szyperski,
*Import is not Inheritance – Why we need both: Modules and Classes*,
Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands, June, 1992, LNCS 615, Springer-Verlag, Berlin, 1992.
[Szyperski–92b] C. A. Szyperski,
*Insight ETHOS – On Object-Orientation in Operating Systems*,
PhD Thesis, Diss. No. 9456, ETH Zürich, 1991.
[Szyperski–96] C. Szyperski,
*Independently Extensible Systems – Software Engineering Potential and Challenges*,
Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, 1996.
[Teitelman–84] W. Teitelman,
*A Tour Through Cedar*,
IEEE Software, April, 1984.
[Templ–94] J. Templ,
*Metaprogramming in Oberon*,
PhD Thesis, Diss. No. 10655, ETH Zürich, 1994.
[Vetterli–91] Ch. Vetterli,
*OPUS: Entwurf und Realisierung eines erweiterbaren, objectorientierten Dokumentenverarbeitungssystems*,
PhD Thesis, Diss. No. 9456, ETH Zürich, 1991.
[Weck–94] W. Weck,
*Putting Icons into (Con-) Text*,
Proceedings of the 13th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS EUROPE '94, Versailles, France, Prentice Hall, 1994.
[WiGu–89] N. Wirth, J. Gutknecht,
*The Oberon System*,
Software – Practice and Experience, 19:9, September 1989.
[WiGu–92] N. Wirth, J. Gutknecht,
*Project Oberon. The Design of an Operating System and Compiler*,
Addison-Wesley, New York, ISBN 0–201–54428–8, 1992.
[Wirth–82] N. Wirth,
*Programming in Modula-2*,
Springer-Verlag, Berlin, 1982.
[Wirth–88] N. Wirth,
*The Programming Language Oberon*,
Software – Practice and Experience, 18:7, pp. 671–690, July 1988.

[Wirth–95] N. Wirth,
*A Plea For Lean Software*,
IEEE Computer, pp. 64–68, February 1995.
[Wolfram–88] S. Wolfram,
*Mathematica, A System for Doing Mathematics by Computer*,
Addison-Wesley, ISBN 0–201–19334–5, 1988.
[Wolfram–93]
*MathLink Reference Guide*,
Wolfram Research, Inc., Mathematica Technical Report, 1993.

# Curriculum Vitae

| | |
|---|---|
| 7 Dec 1964 | Born in Erlangen, Germany |
| 1984 | Abitur at Rudolf Diesel Gymnasium, Augsburg, Germany |
| 1984–1986 | Studies in mathematics at University Augsburg |
| 1986–1990 | Studies and diploma in computer science at ETH Zurich. Diploma thesis: "Support for Remote Procedure Calls (and Persistent Objects) in Oberon" |
| 1990–1996 | Research and teaching assistant at the Institute of Scientific Computing at ETH Zurich |