Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: INF/01

Manuzio: an Object Language for Annotated Text Collections

Marek Maurizio, 955378

Supervisor
Renzo Orsini

PhD Coordinator
Antonio Salibra

January, 2010

Author's Web Page: http://www.dsi.unive.it/ marek/

Author's e-mail: marek@dsi.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

Abstract

More and more large repositories of texts which must be automatically processed represent their content through the use of descriptive markup languages. This method has been diffused by the availability of widely adopted standards like SGML and, later, XML, which made possible the definition of specific formats for many kinds of text, from literary texts (TEI) to web pages (XHTML). The markup approach has, however, several noteworthy shortcomings. First, we can encode easily only texts with a hierarchical structure, then extra-textual information, like metadata, can be tied only to the same structure of the text and must be expressed as strings of the markup language. Third, queries and programs for the retrieval and processing of text must be expressed in terms of languages where every document is represented as a tree of nodes; for this reason, in documents where parallel, overlapping structures exists, the complexity of such programs becomes significantly higher. Consider, for instance, a collection of classical lyrics, with two parallel hierarchies lyric > stanzas > verses > words, and lyric > sentences > words, with title and information about the author for each lyric, and where the text is annotated both with commentary made by different scholars, and with grammatical categories in form of tree-structured data. Such a collection, if represented with markup techniques, would be very complex to create, manage and use, even with sophisticated tools, requiring the development of complex ad-hoc software. To overcome the above limitations due to the use of markup languages partial solutions exist, but at the expense of greatly increasing the complexity of the representation. Moreover, markup query languages need to be extended to take these solutions into consideration, making even more difficult to access and use such textual collections.

In the project "Musisque deoque II. Un archivio digitale dinamico di poesia latina, dalle origini al Rinascimento italiano", sponsored by the Italian MIUR, we have built a model and a language to represent repositories of literary texts with any kind of structure, with multiple and scalable annotations, not limited to textual data, and with a query component useful not only for the retrieval of information, but also for the construction of complex textual analysis applications. This approach fully departs from the markup principles, borrowing many ideas from the object-oriented models currently used in programming languages and database areas. The language (called Manuzio) has been developed to be used in a multi-user system to store persistently digital collections of texts over which queries and programs are

evaluated.

Our model considers the textual information in a dual way: as a formatted sequence of characters, as well as a composition of logical structures called *textual objects*. A *textual object* is a software entity with a state and a behavior. The state defines the precise portion of the text represented by the object, called the *underlying text*, and a set of *properties*, which are either *component* textual objects or *attributes* that can assume values of arbitrary complexity. The behavior is constituted by a collection of local procedures, called *methods*, which define computed properties or perform operations on the object. A textual object T is a *component* of a textual object T' if and only if the underlying text of T is a subtext of the underlying text of T'. The Manuzio model can also represent aggregation of textual objects called *repeated textual objects*. Trough repeated textual objects it is possible to represent complex collections like "all the first words of each poem" or "all the first sentences of the abstracts of each article" in a simple and clean way. A *repeated textual object* is either a special object, called the *empty textual object*, or a set of textual objects of the same type, called its *elements*. Its underlying text is the composition of the underlying text of its elements. Each textual object has a type, which represents a logical entity of the text, such as a word, a paragraph, a sentence, and so on. In the Manuzio model types are organized as a lattice where the greatest element represents the type of the whole collection, and the least is the type of the most basic objects of the schema. Types can also be defined by inheritance, like in object-oriented languages.

Manuzio is a functional, type-safe programming language with specific constructs to interact with persistently stored textual objects. The language has a type system with which to describe schemas and a set of operators which can retrieve textual objects without using any external query language. A persistent collection of documents can be imported in a program and its root element can be referenced by a special variable `collection`. From this value all the textual objects present in the collection can be retrieved through operators that exploit their type's structure: the *get* operator retrieve a specific component of an object, while the *all* operator retrieve recursively all the components and subcomponents of a certain type of an object. Other operators allow the creation of expressions similar to SQL or XQuery FLOWR expressions. Since the queries are an integrated part of the language, they are subject to type-checking and can be used in conjunction with all the other language's features transparently.

The concepts introduced by the thesis have been exploited to develop a prototype of a system where Manuzio programs can be evaluated over a collection of texts. Such system has been successfully used to test and refine our approach.

Acknowledgments

I would like to express my gratitude to my supervisor Renzo Orsini, for his scientific guidance and support during my Ph.D. studies. Thanks to my external reviewers Antonio Albano and Roberto Zicari for the time they spent on carefully reading the thesis and for their useful comments and suggestions. Thanks to the Department of Computer Science of the University Ca' Foscari of Venice for financing my studies with a three years grant, and thanks to all the people working there, in particular Annalisa Bossi and Antonio Salibra for their coordination of the Ph.D. course. A special consideration must be made for Paolo Mastrandrea and the "Musisque deoque II. Un archivio digitale dinamico di poesia latina, dalle origini al Rinascimento italiano" project, sponsored by the Italian MIUR, that introduced me to the problem and provided stimulating examples. Many thanks to my Ph.D. colleagues and to all my long standing friends, especially Camilla. A special thanks goes also to my family for their unconditional support.

Contents

1

Introduction

Research in the field of humanities is often concerned with texts in the form of documents, literary works, transcriptions, dictionaries, and so on. Interpreting, analyzing, sharing insights and results on these documents is the core of scholarship and research in literature, history, philosophy, and other, similar, fields. The recent technology advancements are affecting profoundly the study and use of texts in the humanities area, and a growing number of projects and experiments have been presented in literature to explore the possibilities given by electronic representations of those texts. Such digital collections are considered to be an extremely useful addition to the set of tools available to humanities researchers. Computational methods can provide an overall picture of large quantities of textual material, or particular patterns in the text, that would be difficult to obtain otherwise. These methods, even if advanced, rarely provide final results, but are nevertheless powerful tools in the hand of humanities researchers that can use them to reduce the cognitive gap between physical texts (like books, handwritten documents, and so on) and their digital representation.

The process of digital analysis has been approached firstly in the late 60s, when early ad-hoc systems were used for textual storage and retrieval. In 1986 the Standard Generalized Markup Language (`SGML`) begun to offer a mechanism to define a markup scheme that could handle many different types of text, could deal with metadata, and could represent complex scholarly interpretation along with basic structural features of documents. In 1990 the first edition of the Guidelines for Electronic Text Encoding and Interchange from the Text Encoding Initiative (`TEI`) has been distributed to serve as an `SGML` encoding standard, later converted to `XML`. The `TEI` is often considered the most significant intellectual advance in the field, and has influenced the markup community as a whole. The use of the `XML` data model, however, brings some important drawbacks caused by its tree-like structure. The representation of multiple, concurrent hierarchies, a common situation in literary analysis, becomes a challenge in such context. Multiple solutions has been proposed to overcome this problem and to adapt `XML` query languages, but they are in fact workarounds that substantially store a graph structure in a tree one. At the end of the 90s an important contribution in literature proposed to see text like an "ordered hierarchy of content objects"; a text is viewed as multiple hierarchy

of logically recognizable entities like words, paragraphs, and so on. Other systems employ an ad-hoc data model and their query languages can successfully handle multiple hierarchies. While many are based on `XML`, others depart completely from it or offer a different markup language based on a graph data model as a way to represent textual information.

The Manuzio project aims to be a general tool to be used in the field of literary text analysis. Manuzio offers a powerful, yet simple, formal model to represent complex, concurrent textual structures and their annotations in an efficient way; its underlying data model is in fact a directed acyclic graph. The model is based on the idea of texts as multiple hierarchies of objects: objects are modeled in a way similar to classical object-oriented programming languages objects. Each textual object has a type, and each type has a specific interface that specify which components, attributes, and methods are accessible by the user. Textual objects also support inheritance, so that objects can be declared as members of a subtype of another object's type. All the characteristics of such object are inherited. Textual schemas arises from well-established concepts of the object-oriented programming and databases field. They are instanced through the use of a data definition language that allows users to specify which textual objects types are present in the text, their relationships, and which annotations can be applied to them. In our project, all the documents in a corpus must conform to a certain Manuzio model. The possible specification of constraints directly in the model definition is also being investigated.

Textual objects are stored persistently in what we call a textual database. In our project each textual database represents a self-contained corpus where all documents share the same Manuzio model. This choice makes the implementation of the prototype easier and seems to be the main way in which textual analysis systems are organized. The persistent layer is abstracted from the final user and can accommodate all the possible Manuzio models. The user interacts with textual databases through the Manuzio language, a Turing-complete programming language with persistent capabilities and specific constructs to query, annotate, and explore textual objects and their annotations. From the language point of view the persistency is transparent, so that textual objects are treated just like any other of the language's values.

We developed also a simple but working prototype to test and explore the potentialities of our approach. To easily reach this goal, we designed the prototype in a modular way: the constructs of the Manuzio language are partitioned in self-contained bundles. Each bundle represents a logical portion of the language, like query operators or selection, or a specific type with some associated operators, like integers, records, or textual objects. The Manuzio interpreter has been developed entirely in an object-oriented way so that each element of the language is represented by an object. Bundles are collections of these elements and each one can be included or excluded dynamically when the interpreter is started. We find that this approach gives a high degree of control on the evolution of a language and is, at the

same time, appropriate for the development of a prototype.

The Manuzio language is meant to be used in a system that allows users to perform common queries in a simple, graphical way, but, at the same time, gives the possibility of using more complex interfaces to perform arbitrary complex tasks. The results of such queries can be annotated and shared with other users. Layers of annotations on the same corpus can be created, shared, and eventually merged.

The Manuzio project aims to draw a bridge between humanities researchers and programmers; when analyzing a text the humanities researchers often experience a cognitive distance between their work and the data they work with. When working with `XML`-encoded texts with multiple hierarchies, for instance, the difficulty in expressing queries of arbitrary complexity can hinder the research process. When using ad-hoc, graphical systems, instead, the researcher is limited to the answers the system is meant to give, and to expand such limits is an often difficult task. Through the use of an interactive query language with specific, `SQL`-like, constructs users can express common queries in a simple way. At the same time, however, they are working with a full language that can express algorithms of arbitrary complexity, so that they are not limited by a set of predefined inquiries.

In our opinion a system like the one we just sketched allows a rich environment for humanities researchers that works in the field of literary analysis. The goal of this work is to explore such possibility by defining a formal model for texts, a specification of an ad-hoc programming language, and the sketch of a system where users can interact with such language. We are aware that the project is ambitious and that the development of a completely new programming language is a difficult task. For this reasons the main outcome of the thesis is not a full system, but a prototype that, despite its limitations, will be used to test the feasibility of the whole project. While departing from the *de facto* standard of `XML` can be considered unpropitious, we feel that the representation of concurrent structures in textual data requires the use of a different, graph-based, data model. In our project `XML` still plays an important role as the preferred language to export and share results with other systems, as well as the most probable way of feeding the textual database's parsing process. We also choose to not make use, for now, of other approaches that already exists in literature like other textual data models, specific storage systems, or text retrieval languages. We felt that, in this way, our results would not be biased in a direction or another. We are aware that the development of a system like the one we just described from sketch is a long and difficult task that goes beyond the scope of this work; in the thesis, in fact, we only scratched the surface of our ideal system's features. We feel that, however, a prototype that could be tested by experts of the domain of applications has been an useful guideline to follow in future implementations.

2

Motivations and Literature Survey

"*Talk of nothing but business, and dispatch that business quickly.*" – Aldo Manuzio (on a placard placed in the door of his printing office)

2.1 Motivations and Goals

Literary text analysis is usually carried out through the use of programs written in a general-purpose programming language. In these programs textual data is often stored persistently in `XML`, a meta-markup language that has recently become a de facto standard in data representation and interchange. The data is then accessed from the programming language through the use of `XML` parsing libraries that performs a mapping between `XML` hierarchies and the language's data structures. Despite some standard `DTD`s, like the one proposed by the `TEI` initiative, exist, the encoding schemas are often arbitrary and the details of their structure are left to the single implementations. Moreover, the `XML` markup language is not well-suited to represent the structure of literary texts, since its underlying data model is basically a tree, unable to denote parallel structures. This is not an uncommon requirement in the field of literary analysis: a typical digitalization of a poem, for instance, should provide information about both metrical and prosodic structures, such as verses and sentences, that are not hierarchically nested. Another important shortcoming can be found in annotations, an important feature in programs of textual analysis. In systems that use `XML` as their data model annotations are often stored as simple strings or only basic data types.

Finally, the lack of integration between the data and the programming language do not encourage a clean programming style: the `XML` structures are usually accessed through the use of query languages, like XPath or XQuery, that are embedded in the programming language rather then part of it. The lack of specific constructs to interact with text can also be an obstacle to the production of new queries for both programmers and experts of the domain of application.

While different approaches, that will be discussed in the rest of the chapter, exists, our analysis of such approaches and the exchange of views with experts humanities researchers led us to believe that the field could benefit from the creation

of a general-purpose tool to write complex programs of textual analysis over persistently stored encoded texts where:

1. the data is stored persistently with an encoding system suitable to hold parallel, overlapping, hierarchies;
2. annotations on portions of the text can be of complex type, with the possibility of referencing other portions of the text;
3. a full programming language with persistent capabilities and special textual features is used to define the structure of the data and access it in a transparent, type-safe, way;
4. results of computations can be shared and compared between users.

The Manuzio system, presented in this work, is a proposal of such tool. The goal of Manuzio is to encode literary texts in a model where portions of text with a logical meaning are seen as textual objects, entities with strong similarities to objects of object-oriented language. Textual objects have types, and are organized in a containment relation. Such structure can be defined by an expert humanities researcher through a formal data definition language where concurrent hierarchies and complex annotations can be defined in a straightforward way. Textual objects are stored persistently in a textual repository together their annotations. Each textual repository contains a set of documents with the same structure, created by parsing an input text with a set of recognizer functions, one for each declared textual object type.

The Manuzio programming language, a Turing-complete language with persistent capabilities and specific constructs to deal with textual objects, allows to express textual analysis algorithms of arbitrary complexity. A subset of the language, with a syntax and a semantics similar to the one of other query languages like `SQL`, can also be used to interactively retrieve and annotate specific portions of text without strong programming skills. The language is intended to be used in an environment where textual repositories are available online and users with different access privileges can perform simple queries, launch complex text analysis programs, annotate textual objects, and share their results with other users. Figure 2.1 shows the main components of the system and their users. A domain expert models a collection of textual documents with the model discussed in Chapter 3. The Manuzio system takes the model in input to produce a textual repository. A set of parsing tools are used to load the documents into the repository. The data can then be accessed through programs and queries written in the Manuzio language, either by expert users or by means of a graphical interface.

During the rest of the thesis such components will be discussed in turn and, in Chapter 6, the schema will be revisited to explain in depth how they interact with each other to shape the Manuzio system. In the following sections a brief

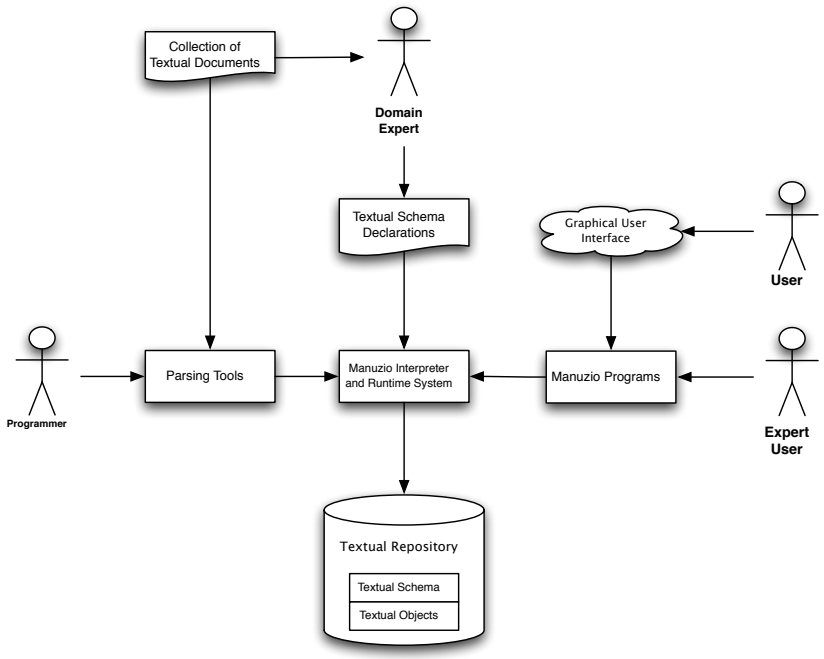

Figure 2.1: The Manuzio system.

history of text encoding will be presented, with particular attention to the problem of overlapping hierarchies, data structure definition, and annotation capabilities of the discussed approaches. After this discussion, in Section 2.4, the main drawbacks of these approaches are put in evidence and we show how our approach, the Manuzio system, can solve the majority of them in a graceful way.

2.2 A brief history of text encoding

In this section a quick overview of text encoding history will be given. While this brief introduction will be useful to understand the motivations of the proposed work and to justify some of our design choices, the reader can see [Hockey, 2004, Schreibman et al., 2004] for a more in depth discussion.

2.2.1 Introduction and Descriptive Markup

The simplest way of representing text in computing is through the use of *plain text*, a sequence of bits that can be interpreted as characters according to some encoding like `ASCII` or *Unicode*. Plain text is the most used way to represent generic texts without attributes such as fonts, subscripts, boldfaces, or any other formatting instruction. All the information that a plain text carries is included in the text itself. The main advantage of plain text is its human readability. A literary text can be represented

by a plain text document, like in the following example that contains the first lines of the *Peer Gynt* from Ibsen.

Example 1 ***(Plain Text)***

```
AASE: Peer, you're lying!
PEER GYNT : No, I'm not!
AASE: Well then, swear to me it's true.
PEER GYNT: Swear? why should I?
AASE: See, you dare not! Every word of it's a lie.
```

Note that extra-textual information, like the name of the speaker, can be included in plain text format by the use of *conventions*. In Example 1, for instance, each line contains a speech and the speaker name is placed at the head of the line, separated from the speech itself by a colon and a space. As the complexity of extra-textual data grows, it can be difficult to represent all such information in a plain text while still maintaining a good degree of readability.

It is common to enrich digital text transpositions with information, distinct from the text, to identify logical or physical features of such text or to give some instructions on how to control text processing. Typically this additional information, called *markup*, is represented as codes or character sequences that intersperses the original text. The term markup is derived from the traditional publishing field where it refer to the practice of "marking up" a manuscript with conventional symbolic printer's instructions in the margins and text of a paper manuscript. Markup was commonly applied by typographers, editors, proofreaders, publishers, and graphic designers.

In the mid-60s the use of computers to compose, print, and typeset text was achieved by marking a binary representation of the text with codes distinguished from the text by special characters or sequence of characters. The file could then be processed and such codes were to be translated in formatting instructions for the printer or phototypesetter that created the final output.

These codes quickly evolved into macros, abbreviations of formatting commands that made easier the work of the compositor. The use of macros gave rise to an interesting question; was the macro simply an abbreviation for a set of formatting instructions, or would it be more natural to see a macro as an identity for a text component such a title, a chapter or an extract? In the first case two different text components that happen to share the same formatting are regarded as equals, but are still different in the second case. This latter approach had a number of advantages. It was possible, for instance, to alter the appearance of a textual component, like a figure caption, without necessarily altering the appearance of any other text component, thus simplifying the compositor work. Moreover, such approach allows documents to be treated as structured data and thus be stored, queried, and analyzed in an efficient way.

Such practice took the name of *descriptive markup* and has since then been regarded as the most natural and correct approach to organize and process text

[Goldfarb and Rubinsky, 1990, Coombs et al., 1987b]. In late 70s the widespread usage of such markup methods gave birth to the first effort to develop a standard machine-readable definition of markup languages, SGML.

2.2.2 SGML

The first effort to produce a text description language standard has been taken in the 80s by the American National Institute (ANSI) and the results of such efforts have been published in 1986 under the name of *Standard Generalized Markup Language* (SGML) [Goldfarb and Rubinsky, 1990].

SGML is a language to create other descriptive markup languages. For this reason a more precise definition of SGML is *metalanguage*, a language for defining other languages. This decision was taken to overcome the difficulty of defining a common markup vocabulary for the entire publishing industry and to make the standard open to new opportunities and innovations. At the same time this approach ensured that any machine capable of recognize and parse SGML would have been capable of handle any SGML compliant language. SGML generates tag languages that use markup to delimitate different portions of text and organize them in a hierarchical way. Elements can have associated *attributes* to specify metadata on them. This logical view of the text was paired with a standard for assigning general formatting instructions to markup tags called *Document Style and Semantics Specification Language* (DSSSL).

In SGML a portion of marked text is called an *element*. The term element can be used in a slightly ambiguous way, sometimes it refers to a specific entity like *a* title or *a* verse, and sometimes it refers to the general kind of object like title or verse. While in natural language such ambiguity is common enough to allow the reader to clearly distinguish between the two different meanings, in SGML there is a formal distinction between the two, so that the latter is called an *element type*.

One of the main notion of SGML is that of *document type*, a specification of a particular set of element types and relationships among them. Examples of document type can be a novel, a letter, a scientific article, a catalogue and so on. An example of relationship can be: an item of a catalogue can occur in a category, or a play has a title and one or more acts, composed by speeches, and so on. Such definitions are not predefined in the SGML standard but are left to the specific language designer. A *Document Type Definition* (DTD) defines a specific document type language in a formal way through a series of markup declarations.

SGML did not have a large scale application in the consumer publishing industry and text processing as it was expected. According to [Schreibman et al., 2004] such lack of success was related to the emergence of *What You See Is What You Get* (WISIWYG) word processing on desktop publishing. Another explanation, given by the same authors, is that SGML specification was too long, had too much optional features, and some of them were too hard to program. For instance the CONCUR feature of the language, that will be discussed more in depth in the next section when we will talk about overlapping hierarchies, while useful for certain applications

were not implemented by all the SGML parsers and thus could not be used outside a controlled environment. One of the main applications of SGML has been its usage for the definition of HTML. However even if successful HTML had flaws, like missing a DTD, that made some of the lacks of the metalanguage clear. Other sources that examine the perspective of using SGML as part of a document management system can be found in [Alschuler, 1995, Travis and Waldt, 1995].

2.2.3 XML

XML, Extensible Markup Language, is a standard for document markup. It defines a generic syntax used to markup data with human-readable tags and aims to provides a standard format for computer documents. Data is included in XML as strings of text surrounded by markup that describes the data and its properties. XML's basic unit of data is called an *element*. The XML specification defines the exact syntax that the markup must follow: how elements are delimited by tags, which characters are used to define tags, what names are valid element's names, how to specify attributes, and so on. XML is a *metamarkup language*, it does not have a fixed set of tags and elements but, instead, allows developers to define the elements they need. In XML the tags structure must be strictly hierarchical, but few other restrictions apply.

Restrictions on the permitted markup in a particular XML document can be defined in a *schema*. Document instances can be compared to the schema: documents that match the schema are said to be *valid*, while documents that does not match are said to be *invalid*. There are several different XML schema languages, with different levels of expressivity, the most broadly supported is the document type definition (DTD). All current schema languages are declarative, thus there are always some constraints that cannot be expressed without a Turing-complete programming language.

XML is a descendant of SGML and it has been of immediate success from the time the first specification was born in 1998. Many developers who needed a structural markup language but hadn't been able to bring themselves to accept SGML's complexity adopted XML. It has since then used in the most different contexts. In the course of years other, related, standards arose, like the Extensible Stylesheet Language (XSL), an XML application for transforming XML documents into a form that could be viewed in web browsers. This technology soon become XSLT, a general-purpose language for transforming one XML document into another. The Cascading Style Sheets (CSS) language, already in use in the HTML standard, were adapted by the W3C to have the explicit goal of styling XML documents to be displayed in a browser.

XML is essentially a simpler, more constrained version of SGML, so that it would be easier to develop applications based on XML languages. The original specification was only 25 pages long instead of the 155 of SGML. The main difference of XML is that it allows the creation of new markup languages without the need to specify a DTD for them. In SGML, for instance, a closing tag could be omitted; without a

document definition such omission could generate ambiguity in many cases. In XML the mandatory closing tag makes every well-formed document sound. If an XML document is valid its semantics can be understood by a parser unambiguously even without a DTD. We will see that each XML document is essentially a hierarchy (or tree) of elements with attributes. Even if it is not strictly required an XML document can have an associated document type definition to which it must conform.

Source Code 1 An example of XML text encoding: dramatical hierarchy.

```
<document>
        <play title="Peer Gynt" author="Henrik Ibsen">

                ...

                <sp who='Aase'> Peer, you're lying! </sp>
                <sp who='Peer'> No, I'm not! </sp>
                <sp who='Aase'> Well then, swear to me
                                it's true! </sp>
                <sp who='Peer'> Swear? why should I? </sp>
                <sp who='Aase'> See, you dare not!
                                Every word of it's a lie!
                </sp>

                ...

        </play>

        ...

</document>
```

Source Code 2 An example of XML text encoding: metrical hierarchy.

```
<document>
        <play title="Peer Gynt" author="Henrik Ibsen">

                ...

                <l> Peer, you're lying!  No, I'm not! </l>
                <l> Well then, swear to me it's true! </l>
                <l> Swear? why should I?  See, you dare not! </l>
                <l> Every word of it's a lie! </l>

                ...

        </play>

        ...

</document>
```

While this quick introduction is sufficient for the purpose of the thesis further informations on XML metalanguage can be found in [Bray et al., 2000]. In Source Code 1 the Peer Gynt text presented earlier has been encoded in XML. Other parts

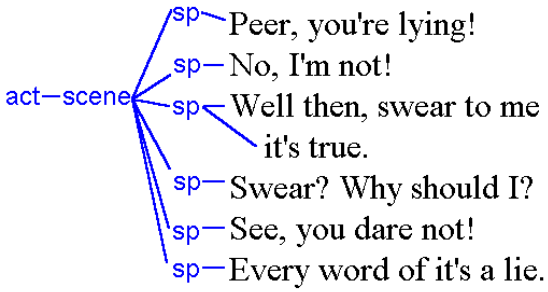

Figure 2.2: The graphical representation of dramatic structure.

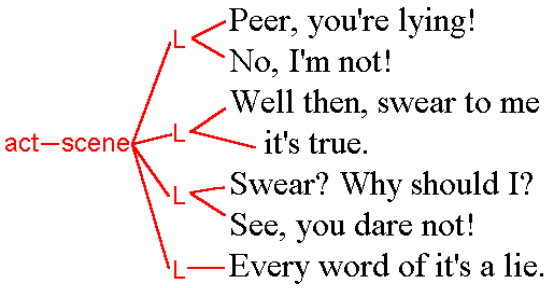

Figure 2.3: The graphical representation of metrical structure.

of the document have been omitted and replaced by dots. Extra textual information is encoded as attributes, while the XML elements represents the play's speech structure. In Source Code 2, instead, the metrical structure is represented. Both these structures are in fact trees, as shown in Figure 2.2 and Figure 2.3. However, since in a well formed XML document's tags cannot overlap, it is difficult to represent both metrical and dramatic structures at the same time. For further information on the XML metalanguage, see [Light, 1997, Connolly, 1997]. A plethora of Web-based material is present, among which the World Wide Web Consortium (W3C) pages that provides official documentation on XML as well as SGML.

2.2.4 The Text Encoding Initiative

The first encoding of a literary text in digital format is attributed to Father Roberto Busa that encoded in IBM punched cards the *Index Thomisticus* in 1949. Similar works proliferated and in 1967 an already long list of "Literary Works in Machine Readable Format" were published [Carlson, 1967]. Since then a standard encoding format has been pursued, as it would have brought a number of obvious advantages in data sharing and results access. Hindering such achievement were mainly the non-trivial differences among the texts to be encoded and the difficulty to define a single schema to accommodate all of them. Experts felt that the proliferation of different, poorly designed, encoding systems would have been a treat to the computer support in humanities research.

To solve this problem the *Text Encoding Initiative* (TEI) [Sperberg-McQueen et al., 1994, Burnard and Sperberg-McQueen, 2005] was founded in the late 80s. Its goal was the definition of a standard encoding for literary texts. The first draft of their guidelines were published in 1990 and is, at the present time, one of the most extensively used encoding systems in humanities applications. Since 2005, the guidelines are released under GNU Public license and the development is taking place in public.

The TEI encoding is based on some principles that have been formulated in the course of its evolution. In particular, in the P1 document the main design goals are:

> The following design goals are to govern the choices to be made by the working committees in drafting the guidelines. Higher-ranked goals should count more than lower-ranked goals. The guidelines should
>
> - suffice to represent the textual features needed for research
> - be simple, clear, and concrete
> - be easy for researchers to use without special-purpose software
> - allow the rigorous definition and efficient processing of texts
> - provide for user-defined extensions
> - conform to existing and emergent standards

The first choice of the guidelines has been the adoption of SGML as a metalanguage for the encoding schema definition, language that has been later replaced by XML. The TEI schema has been formalized in a DTD, a descriptive grammar of documents in the humanities research field. The TEI schema reflect, in general, the main idea of declarative markup languages, so that the emphasis is on logical and abstract structures of the text rather then on physical features. However, the markup can be used also in a presentational way, by marking physical structures, when a logical interpretation is not practicable or the research methods employed are based on a physical description of the document.

The textual features set of the TEI guidelines are partitioned in three levels: the first is made up of textual elements that are considered universally valid for any kind of document in the domain; the second includes all the elements that are proper of a macro-subset of texts, like prose, poems, drama, dictionaries, and so on; the third set is made of elements that are specific for certain analysis procedures.

This stratified ontology corresponds to a modular structure of the DTD, that has been divided in *tag sets* so that the *core tag set* contains the universal elements, the *base tag set* contains five subsets of elements, one for each of the basic document types, and, finally, the *additional tag set* contains all the document-specific elements. Every TEI user can customize his DTD by combining the core set with one of the basic sets and any number of additional elements. By the use of such modularity, there is no need of an antecedent agreement on what features of a text are important, since such decision is largely left to the encoder. In practice, the TEI defines about 400 elements, but the complexity of schemas is usually kept tractable by the flexibility of the specification.

2.2.5 Text as an OHCO

The success of descriptive markup suggested that it could be more than a way of working with text, it could be the foundation of a digital text model. This idea has been presented in [DeRose et al., 1997] under the label of "what is text, really?". The general idea behind such model is that text consists of objects of a certain sort, structured in a certain way. The nature of objects is suggested by the logical decomposition of a text in, for instance, chapters, sections, paragraphs, sentences and so on, and not by a typographical decomposition in pages, lines etc.

Such objects are organized according to some hierarchical relation, they are contained and contains other objects, and are ordered linearly by the natural linear order of the text they represent. With such assumptions text can be seen as an "Ordered Hierarchy of Content Objects" (OCHO). The OCHO model performs a basic kind of data abstraction over a text, where every portion of text can be abstracted in an object with some fixed properties. The containment relation partially constrain the structure of the text as, for instance, a chapter cannot occur inside a paragraph. Other models of text have been compared to the OCHO like treating the text as a sequence of ASCII characters or graphical glyphs. According to

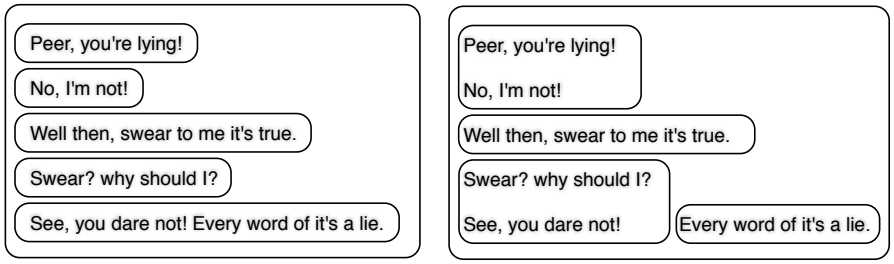

Figure 2.4: An example of content objects.

[DeRose et al., 1997, Coombs et al., 1987b] such model are inferior in functionality and can be automatically generated from an OCHO encoding of the text. Moreover such model copes well with formal languages like SGML or XML.

In Figure 2.4 an example of content objects is given. On the left a dialogue have been split in speeches, on the right the same dialogue is split in metrical lines of verse. The dialogue of the previous section example has been stripped of extra textual information and content objects have been enclosed in boxes. On the left the full dialogue is an object, together with each line of speech. On the right, the dialogue is marked again but this time the verses have been enclosed in boxes. For clearness reasons we omitted to enclose each single word in a box. It is easy to see how content objects can be put in a containment relation: the dialogue contains both verses and speeches, that contains words. The reader will also notice that lines and speeches structures are parallel, and that it is not possible to establish a containment relation between them. This problem, called *overlap*, will be discussed in depth in Section 2.3.

The OCHO view of the text will be of central importance in the development of the Manuzio model, since our approach also see the text as a hierarchy of objects similar to the objects typical of object-oriented languages.

2.2.6 TexMECS and GODDAG

The approach presented in this section is part of the MLCD (*Markup Languages for Complex Documents*) project, that aims to integrate a novel notation, a data structure, and a constraint language to overcome the overlapping problem. One of the MLCD achievements is the specification of the GODDAG (*Generalized Ordered-Descendant Directed Acyclic Graph*) data structure [Sperberg-McQueen and Huitfeldt, 2004], based on the idea that overlaps can be represented as multiple parentage between text nodes. A GODDAG is a directed acyclic graph where each node is either a leaf node, labeled with a string, or a nonterminal node, labeled with a generic identifier. Directed arcs connect nonterminal nodes with each other and with leaf nodes. No node dominate another node both directly an indirectly, but any node can be dominated by any number of nodes.

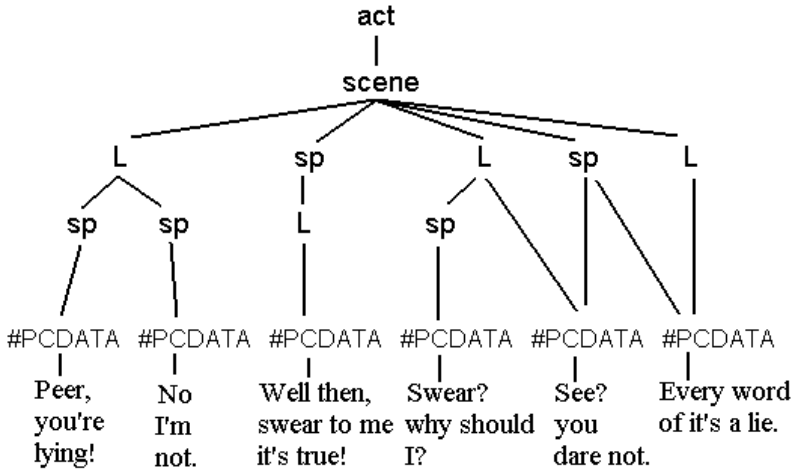

Figure 2.5: Peer Gynt represented as a GODDAG data structure.

In Figure 2.5 the GODDAG data model has been applied to the Peer Gynt example and the results are shown as a graph.

The GODDAG data structure can be *generalized* or *restricted*. The former is a complex data structure that lend itself to the representation of complex documents with multiple roots, alternate text ordering, fragmented elements, and so on. The latter is simpler, often sufficient for literary text representation of documents with concurrent hierarchies and arbitrarily overlapping elements, but without the complex features exemplified above.

The XML data model can be seen as a subset of GODDAGs, since trees are a subset of directed acyclic graphs. This means that it is possible to construct a GODDAG data structure from any XML document. This achievement is very important because it ensure backward compatibility with existing XML corpora. It is also possible, through specific algorithms, to construct GODDAGs from multiple hierarchy documents encoded in XML by one of the techniques presented later in thic chapter.

To denote in a natural and simple way a GODDAG structure a language alternative to XML, called TexMECS has been developed [Huitfeldt and Sperberg-McQueen, 2001]. TexMECS is a markup language intended for experimental work in dealing with complex documents. Every TexMECS document is directly mapped to a GODDAG structure and vice versa. One of its main characteristics is its isomorphism with XML for documents that do not exhibit multiple hierarchies in their structure. Specific algorithms exists to translate XML conventions to represent complex structures to TexMECS and vice versa [Durusau and O'Donnell, 2002].

In Source Code 3 a simple example of TexMECS syntax is presented. The reader can see that the syntax is pretty clear and the human readability of the document is almost the same as plain XML. Since TexMECS is a non-XML syntax, it cannot take advantage of existing XML software and infrastructure, nor of its wide adoption as a

Source Code 3 `TexMECS` encoding of Peer Gynt example.

```
1 <sp who="AASE"|<l|Peer, you're lying!|sp>
2 <sp who="PEER GYNT"|
3 |-l><+l|No, I'm not!|l>|sp>
4 <sp who="AASE"|<l|Well then, swear to me it's true.|l>|sp>
5 <sp who="PEER GYNT"|<l|Swear? why should I?|sp>
6 <sp who="AASE"|See, you dare not!|l>
7 <l|Every word of it's a lie.|l>|sp>
```

standard.

2.2.7 LMNL

`LMNL` (*Layered Markup and aNnotation Language*) is a non-xml approach to model textual data that has been presented in 2002. The main contribution of `LMNL` is its abstract data model:"LMNL documents contain character data which is marked up using named and occasionally overlapping ranges. Ranges can have annotations, which can themselves be annotated and can have structured content. To support authoring, especially collaborative authoring, markup is namespaced and divided into layers, which might reflect different views on the text"[Tennison and Piez, 2002].

The main idea behind the `LMNL` data model is the concept of *range*. The basic form of a range is a contiguous sequence of characters of the base text, enclosed by named tags that indicates the range start and end. Unlike `XML` ranges can overlap freely. A document consisting of just text and ranges are called a *flat* document, a document where exists just two *layers*, the text layer and the range layer. `LMNL` can extend flat documents with additional layers to define complex relationship between ranges in two way. First, other layers that contains ranges built over the text can be added. This gives a completely alternative view of the document where ranges of different layers are not related to each other. Second, layers can contains ranges that range over other ranges instead that over the base text. With this feature it is possible to create complex relationships between layers. Each range can be annotated with a string, a simple annotation, or with a portion of `LMNL` code, a complex, structured, annotation.

The example in Source Code 4 shows the first lines of Peer Gynt encoded in `LMNL`. The reader can easily understand the syntax, where the speaker is denoted as an attribute of the *speaker* element. An example of structured annotations can be seen in the author name of the example, where a simple first name and last name structure has been adopted.

The example shown above uses a `LMNL` ad-hoc syntax. Other encoding syntaxes have been proposed: one of them, called `CLIX` (*Canonical LMNL In XML*) is a way to encode the `LMNL` abstract data model into a plain `XML` document. Presented in [DeRose, 2004], `CLIX` is based on the work done with `OSIS` (*Open Scripture Information Standard*), this approach main idea is is to flatten a `LMNL` document into an

Source Code 4 The Peer Gynt example encoded in LMNL.

```
[document}
  [play [author [firstname}Henrik{firstname] [lastname}Ibsen{lastname]]}
    ...
    [speaker [who}AASE{who]}[line}Peer, you're lying!{speaker]
    [speaker [who}PEER GYNT{who]No, I'm not!{speaker]{line]
    [speaker [who}AASE{who][line}Well then, swear to me it's true!{line]{speaker]
    [speaker [who}PEER GYNT{who][line}Swear? why should I?{speaker]
    [speaker [who}AASE{who]See, you date not!{line]
    [line}Every word of it's a lie!{line]{speaker]
    ...
  {play]
{document]
```

XML document encoded with a sort of milestones called *Trojan Milestones*.

LMNL is an emerging data model that can represent textual information in a natural way. While efficient implementations are still under development, the possibility to represent a LMNL model in XML adds an high degree of portability to such solution.

2.2.8 Textual Models

Textual models focus on the representation of textual data. The goal of such models is to describe texts by their structure, to allow operations of different nature on the texts and to express constraint both structure and operations. In this section two different textual models will be presented, TOMS and MdF. Both shares with Manuzio the ability to represent textual information without being constrained by markup language limitations but, on the other hand, they require an ad-hoc way to store and retrieve data.

Textual Object Management System

The TOMS (*Textual Object Management System*) model has been developed in early 90s [Deerwester et al., 1992] and has been used as part of a full text retrieval system.

TOMS deals with typed *textual objects*, a component of logical interest in the text that can be recognized in some way. The textual model of TOMS consists in a set of textual object types which relationships are defined by a *grammar* that can express the relationships between objects. The possible relationships are:

- Repetition: a list of objects of the same type. For instance a paragraph can be seen as a repetition of sentences.
- Choice: disjoint union of objects. For instance a chapter can be made by either sections or tables.
- Sequence: cartesian product of objects. A fixed sequence of different objects. An email, for instance, can be seen as a sequence of a subject and a body.

- Parallel: union of objects. A set of parallel objects that occurs together in the text. For instance a Bible chapter can be sees as composed of paragraphs *and* of verses at the same time, thus allowing multiple views of the text.

The actual object recognition occurs from a text written in natural language and encoded as plain text through a set of functions called the *recognizer functions*. A recognizer function is dedicated to a specific type; it accepts a string of characters and returns a list of pairs in the form $(index, offset)$ of objects of that type present in the text. The combination of the recognizer functions for all the types in the grammar is called a *document parser*. Recognizer functions are partitioned in three macro-categories:

- Regular Expressions: the `TOMS` system have a built-in regular expressions engine that permits to easily identify all the textual objects that can be recognized by pattern matching.
- Internal: complex recognizer functions can be written in the C programming language and integrated into the `TOMS` function's library.
- Enumeration: a workaround useful for textual objects that are difficult to recognize algorithmically. This kind of functions allows manual specification of the indexes that can be the results of human interaction, previous markup, external recognizers, and so on.

Source Code 5 The `TOMS` structure of the Peer Gynt example.

```
1 document [poem
2   SEQ (
3     title [word],
4     body PAR(
5       [line [word]],
6       [speech [word]]
7     )
8   )
9 ]
```

The Source Code 5 shows how the Peer Gynt example textual objects can be put in relation to form a `TOMS` structure. In this simple example we view the corpus composed of a repetition of poems, each one composed by a title and a body. The body is a parallel structure composed both of a repetition of lines and a repetition of speeches, both composed of words. The syntax used is the one proposed in [Deerwester et al., 1992] where SEQ and PAR represent, receptively, the sequence and the parallel structuring constructs. The REP construct is, instead, abbreviated by square brackets. The CHO construct for disjoint union is not used since it will not have a correspondent in Manuzio.

TOMS is better described as an indexing toolkit, a series of algorithms that create an index of the logical objects of the text, with the word as the fixed basic, indivisible object. A TOMS model is constrained by a grammar that permits to express basic assumption on the text structure. It is not possible, for instance, to constrain the cardinality of repetitions, while it is possible, through the parallel grammar construct, to have concurrent elements, even at root level. The TOMS object model is a *C* data structure that can be interrogated by a *C* library The document hierarchy can be traversed starting from a predefined *root cursor*, a pointer to the top node of the hierarchy. Finally, TOMS also allows for the definition attributes related to objects. Such attributes, however, are not structured and cannot be queried easily as the main text.

The TOMS system was implemented in C at first, but a more flexible version has been subsequently written as an extension of a Perl interpreter, where each TOMS function appears as a Perl primitive.

The TOMS introduces the important concept of basic, indivisible object. However this basic object type is fixed, while a more flexible model should allow the encoder to freely define such type. Another interesting concept of this system is the definition of the root cursor as an entry point to traverse the hierarchy of objects. Both these concepts will be present, with some improvements, in the Manuzio model presented in section 3.

Mdf Model

The MdF (*Monad dot Features*) has been developed in [Doedens, 1994]. It is a database oriented model well suited to store textual data and information about that text. MdF offers an high level view of the underlying database and defines also an access language to query such high level structure. The model has subsequently been refined in [Petersen, 2002, Petersen, 1999].

The MdF model conforms to thirteen requirements described in [Petersen, 2002]:

- Objects: the model must be able to identify separate parts of the text, called objects.
- Objects are unique: each object must be uniquely identifiable.
- Objects are independent: each object in the database must exists without the need of referencing other objects.
- Object types: objects with similar, distinguishable characteristics must be partitioned in types. Most of the times a type identify a logical, human recognizable, part of the text like a chapter or a paragraph.
- Multiple hierarchies: objects types can be organized in multiple, different hierarchies.

- Hierarchies can share types: an object type can be part of one or more hierarchies.
- Object features: objects can have attributes, called features in the model, that represent addition information not present in the text tied to a specific object.
- Variations: the model must be able to accommodate for word variations.
- Overlapping objects: object can overlap, even if they have the same type.
- Gaps: the text identified by an object is not required to be a contiguous portion of the base text.
- Type language: the model needs a type language to specify object types and their features.
- Data language: the model needs a strongly typed data language in which it can specify its `CRUD` operations.
- Structural relations between types: objects of different types can be in relation with each other. The model should provide a language to specify such relations.

The `MdF` model satisfies the first ten requirements. It defines a model for text and a query language, but it does not provides languages to constraint the structure of the objects, to insert, change or delete objects in an high level way, not to specify how objects are structured.

The `MdF` model is based on the concept of *monad*. Monads are the basic building blocks of the database and are represented by integer numbers. An *object* is a set of monads. Every object has an *object type* that determines what features an object has. A *feature* is a function that takes an object as parameter and returns another value.

The backbone of the `MdF` model is a linear sequence of atomic elements called monads. A monad is an entity that can be represented simply by an integer, so that their associated integer number carries their relative ordering and that it is possible to apply standard arithmetic relational operations to them to check for equality or inequality. Note that monads are not text, just integers. The text will be tied to monad through a special object, described later, which instances are composed of just one monad. This high level view of the model data means that the `MdF` model can be successfully applied not only to text but to anything that is strictly linear in nature like, for instance, `DNA` sequences.

An object in a `MdF` model is a set of monads. There are no particular restrictions on this set, so that objects can, for instance, have gaps by being composed by non contiguous monads. Objects are grouped in types, with the only constraint that no two objects of the same type can be composed of exactly the same set of monads. This constraint is needed to satisfy the object identity principle described earlier in

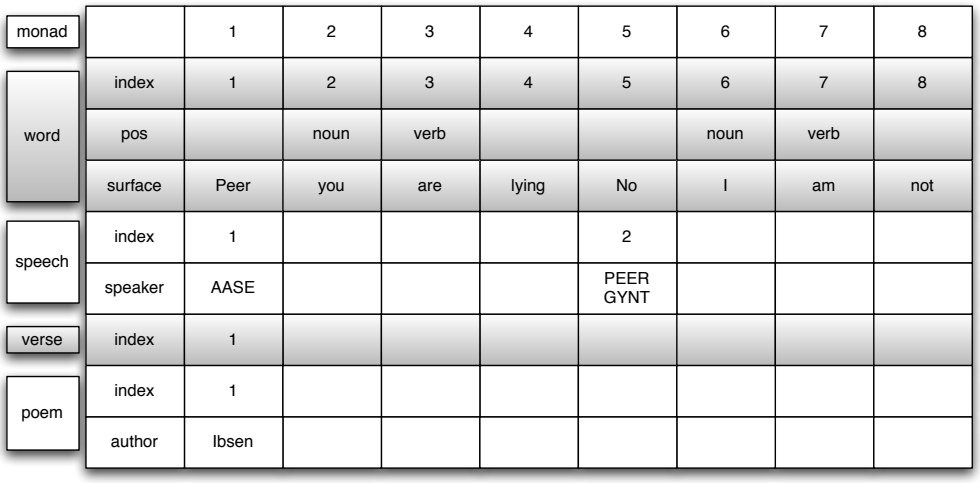

Figure 2.6: Peer Gynt represented as a `MdF` data structure.

the section. On the other hand, two objects of different type can be composed of the same monads without breaking the model rules. It is important to note that, in `MdF` objects are not composed by other objects, but by monads. Relations between objects can be computed by computing basic set operations between their monad sets and by taking the monads ordering into account.

Features are functions from objects to values. The object type of an object determines which features that object has. The domain of a feature is always the set of objects of the object type that define it, while the codomain is unrestricted and can be of any type. In `MdF` features can be partial functions, so that a feature value can be undefined for some objects even if their type specify it. In the `MdF` model the text is modeled by defining an object type, usually `Word`, which serves as the basic type of the model. Every object of this type are composed by exactly one monad and have a special feature, called *surface*, that contains a string representing a portion of the *base text*. Since monads are ordered following the base text linear ordering the union of the basic object surfaces will reconstruct the parts of the text that the model takes into account for analysis.

The `EMdF` (*Extended Monad dot Features*) model, developed in [Petersen, 1999], is an extension of the `MdF` model. It defines and names a set of concepts derived from the four `Mdf` basic concepts. Among them the concept of *all_m* and *any_m* have been of particular interest for our work. *all_m* is the type which have just one object: the one consisting of all monads in the database. It represents the full base text. *any_m* is the object type that we called the base type in the previous paragraph. Every object of that type is composed by exactly one monad.

A graphical representation of an `MdF` database that model the first verse of the Peer Gynt example can be found in Figure 2.6. Each monad is simply an integer number and have a corresponding object of the minimal type *word*. Each word has

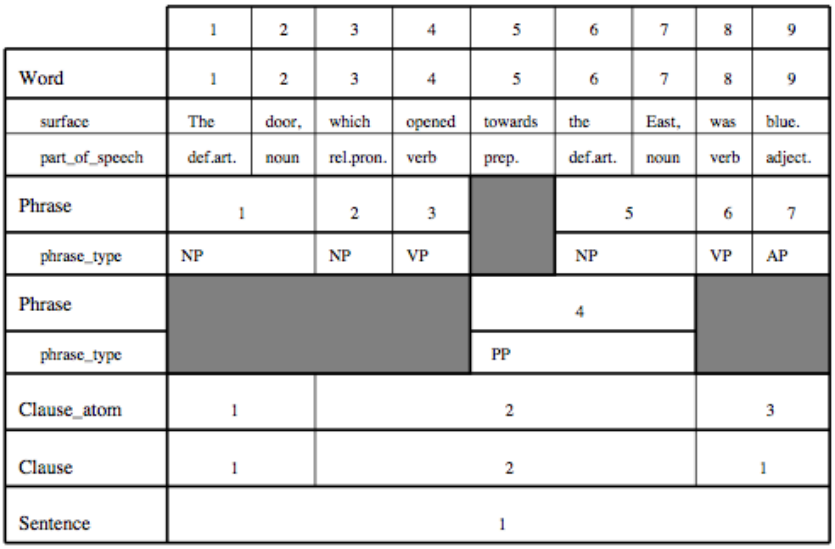

Figure 2.7: MdF example.

a surface feature to store it's corresponding text. Other objects are composed of monads and have features but no text on their own. The full potential of an `MdF` database can be better explained, however, by the example that the author gives in [Petersen, 2002], proposed here in Figure 2.7. Here the modeled textual objects can have gaps and self-overlapping, as can be seen in the *phrase* object of the figure.

The `MdF` model has been of great inspiration to Manuzio thanks to its clean simplicity and its sound mathematical foundations. The concept of maximal and minimal objects, as well as that of object types, will be present in Manuzio as well. On the other hand, differently from `MdF`, objects in Manuzio are composed by other objects. This makes `MdF` best suited for complex, semi-structured hierarchies, text criticism applications, and so on, while our model will be best suited for highly structured text, making its query and constraint languages will be easier to write and use.

2.3 Overlapping Hierarchies

Markup languages have a fundamental shortcoming when dealing with literary texts. They allow the representation of exactly one hierarchy: one single structure of the document can be modeled over a text. `XML`-based markup, for instance, requires that a document is organized hierarchically as a single tree, where each fragment of text is contained in exactly one element and has only one parent.

In some cases this shortcoming is avoidable. For instance the logical structure of a text, i.e. the decomposition in captions, lists, sections etc., differs completely

from the syntactic structure such as sentences and phrases, but none of this element types overlaps, so it is possible to represent them in a single hierarchical structure. However the result is not only an admixture of elements taken from different contexts, but ignoring the problem also induces a containment relation between element types that is not necessarily present in the text.

There are, in literature, plenty of examples in which the same fragment of text needs to be associated to different, possibly hierarchically incompatible, marking. For instance in verse dramas the metrical structure and the dramatic structure are not only different in nature, but can generate what is referred, in literature, as an *overlapping problem* [DeRose, 2004], for instance where a sentence goes beyond the boundaries of a metrical line[1].

Source Code 6 Overlapping Structures.

```
<document>
        <play title="Peer Gynt" author="Henrik Ibsen">

                ...

                <l>
                <sp who='Aase'> Peer, you're lying! </sp>
                <sp who='Peer'> No, I'm not! </sp>
                </l>
                <l>
                <sp who='Aase'> Well then, swear to me
                                it's true! </sp>
                </l>
                <l>
                <sp who='Peer'> Swear? why should I? </sp>
                <sp who='Aase'> See, you dare not!
                </l>
                <l>
                            Every word of it's a lie!
                </l>
                </sp>

                ...

        </play>

        ...

</document>
```

In the Source Code 6 and Figure 2.8 both metrical and dramatic hierarchies are encoded. This code fragment, however, is not a well-formed `XML` document since the `<l>` and `<sp>` tags are not hierarchically nested. While an invalid `XML` document has no meaning to a machine, it is easy to understand intuitively that the meaning of such notation is that a part of the `<l>` element overlaps with a part of the `<sp>` element. In the next section this problem, called *overlapping* of elements, is discussed and different approaches to overcome it are overviewed. We will also see,

[1]Such situation is common in poetry and is called *enjambment*[Abrams and Harpham, 2008]

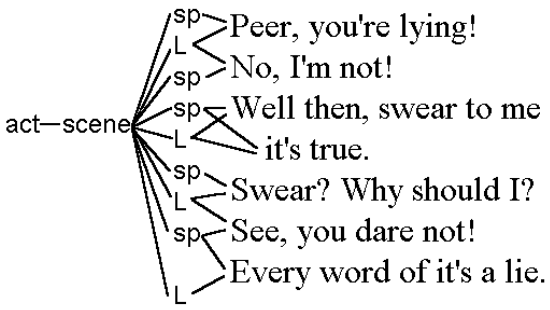

Figure 2.8: The graphical representation of both dramatic and metrical structures.

in later sections, that by using an ad-hoc model for text, like the Manuzio model, is an elegant an efficient way to solve overlaps.

2.3.1 Solutions to the Overlapping Problem

The overlapping problem has been largely discussed in literature. Different ways to mark concurrent elements have been proposed: some o them are special kind of `XML` or `SGML` encoding, while others, like `TexMECS`, `LMNL`, or ah-hoc textual models, relay on a different data model to totally avoid the problem. The choice of using an `XML` based solution to model overlapping hierarchies is mainly dictated by the view of `XML` as a standard for data interchange. In a research environment the use of standards is important so that different groups can work on the same data set, results can be shared, and so on. In the rest of the section the main techniques used to overcome the overlapping problem with `SGML` and `XML` are presented and briefly discussed. A far more complete discussion can be found in [DeRose, 2004].

- *CONCUR*: an optional feature of `SGML` that allows multiple hierarchies to be marked concurrently in a document. Since the definition of this feature was optional in `SGML` it is not implemented by all `SGML`-languages parsers, making its use architecture dependent and thus unreliable. The `CONCUR` feature have not been ported to standard `XML`, even if some attempts to include this option to vanilla `XML` have been proposed [Hilbert et al., 2005]. In Source Code 7 a `CONCUR` example is given.

- *Milestone Elements*: this technique is described in depth in the `TEI` guidelines [Sperberg-McQueen et al., 1994]. In order to encode overlapping structures the milestones approach is to represent one hierarchy as standard `XML`, and the others with empty elements that are seen as virtual starting and ending tags. Such empty elements are called *milestones* and gives the name to the technique. One of the main problem of this approach is the need to define a primary hierarchy. An example of such approach is given in Source Code 8.

A variant, called *flat milestones*, overcome this problem by marking all hierarchies with milestones, but, on the other hand, produces very complex XML documents.

- *Fragmentation of Items*: also called *partial elements* sometimes, is another technique described in the TEI guidelines [Sperberg-McQueen et al., 1994]. Whenever an element overlaps another element, the former is split into two separated elements in a way that preserves the document well-formedness. Syntactic conventions are used to identify partial and non-partial elements and to recognize a fragmented element pieces, as shown in Source Code 9. While this technique allows advanced behaviors like the creation of non-contiguous elements, it also introduces a dominance relationship between different hierarchies that could not be meant by the encoder. Another way to express split elements aggregation is also defined in the TEI specifications under the name of *reconstruction of virtual elements*.

- *Redundant Encoding*: one of the simples approaches to solve the overlapping problem is to have multiple copies of the document, each marked up with a different hierarchy. While no special tools are required to handle the documents, each hierarchy has to be handled separately, so that relationships between elements of different hierarchies cannot be examined. Moreover, multiple copies have to be maintained, a practice that require additional space and is error prone.

Source Code 7 Overlapping hierarchies: SGML concur.

```
<div1 type="act">
<(D,V)div2 type="scene">
<(V)l>
<(D)sp who='Aase'> Peer, you're lying! </(D)sp>
<(D)sp who='Peer'> No, I'm not! </(D)sp>
</(V)l>
<(V)l>
<(D)sp who='Aase'>
                   Well then, swear to me
                   it's true! </(D)sp>
</(V)l>
<(V)l>
<(D)sp who='Peer'> Swear? why should I? </(D)sp>
<(D)sp who='Aase'> See, you dare not!
</(V)l>
<(V)l>             Every word of it's a lie!</(D)sp>
</(V)l>
...
</(D,V)div2>
</div1>
```

With the exception of CONCUR all these approaches are in fact workarounds. They complicate the document compromising human readability and requires special tools

Source Code 8 Overlapping hierarchies: milestones in XML.

```
<sp who='Aase'><lb n="1" />
  Peer, you're lying! </sp>
<sp who='Peer'>
  No, I'm not! </sp>
<sp who='Aase'><lb n="2" />
  Well then, swear to me it's true! </sp>
<sp who='Peer'><lb n="3" />
  Swear? why should I? </sp>
<sp who='Aase'>
  See, you dare not!
<lb n="4" />
  Every word of it's a lie!
</sp>
```

Source Code 9 Overlapping hierarchies: fragmentation of XML elements.

```
<sp who="Aase">
  <l part="i">Peer, you're lying!</l>
</sp>
<sp who="Peer">
  <stage>without stopping</stage>
  <l part="f">No, I'm not!</l>
</sp>
<sp who="Aase">
  <l part="n">Well then, swear to me it's true!</l>
</sp>
<sp who="Peer">
  <l part="i">Swear? Why should I?</l>
</sp>
<sp who="Aase">
  <l part="f">See, you dare not!</l>
  <l part="n">Every word of it's a lie.</l>
</sp>
```

to be handled. Standard query languages for XML like, for instance, *XPath* and *XQuery* are not natively compatible with such approaches[2].

Stand-off Markup

Another technique, initially defined in the TEI guidelines [Sperberg-McQueen et al., 1994], is the stand-off markup technique that has later been widely adopted by various projects as a way to represent multiple, concurrent hierarchies of a single document. The main idea is to have a source document written in XML or in plain text, and a number of external documents, written in XML, where each element references a portion of the source document[3]. Each external document is a view of the source and represents an independent hierarchy. If the source document is plain text then all hierarchies have equal importance, but we can define a main hierarchy by includ-

[2]There are, however, specialized versions of such languages that takes multiple hierarchies into account [Jagadish et al., 2004, Witt, 2004a], but this goes beyond the scope of this thesis.

[3]While any sufficiently powerful pointing mechanism will work the TEI suggest the usage of *XPointer*. Using a powerful tool allows also the creation of non-contiguous elements.

ing it in the main document without losing the possibility of working with external documents. The creation of unwanted dominance relationships between hierarchies is thus avoided.

In the example in Source Code 10 the Peer Gynt metric and dramatic structures are encoded using a stand-off markup. In the main document the dramatical hierarchy is encoded in standard XML, together with other elements that represents line fragments. In the external document presented in Source Code 11 the metrical structure is reconstructed through the TEI *join* element.

Source Code 10 Standoff markup example, the main document contains the dramatical structure.

```
<sp who="Aase">
  <l id="L1a">Peer, you're lying!</l>
</sp>
<sp who="Peer">
  <stage>without stopping</stage>
  <l id="L1b">No, I'm not!</l>
</sp>
<sp who="Aase">
  <l id="L2">Well then, swear to me it's true!</l>
</sp>
<sp who="Peer">
  <l id="L3a">Swear? Why should I?</l>
</sp>
<sp who="Aase">
  <l id="L3b">See, you dare not!</l>
  <l id="L4" >Every word of it's a lie.</l>
</sp>
```

Source Code 11 Standoff markup example, the external document realize the metrical structure.

```
<joinGrp result="l" targOrder="y" targType="L">
<join scope="branches" targets="L1a L1b"/>
<join scope="branches" targets="L3a L3b"/>
</joinGrp>
```

The main advantage of stand-off markup is the capability of working on read only documents. In this way a read-only corpus can be analyzed or annotated by different teams in different ways with ease. While powerful the stand-off markup approach completely destroy the XML documents human readability, and the resulting documents cannot be queried, edited or validated without specific, ad-hoc, tools.

2.4 Conclusions and Comparison

In this chapter a survey of existing approaches to the digital representation and analysis of literary texts has been given. While the topic is broad, each solution has been presented from an high-level point of view to emphasize the need of other,

more specific tools to define textual schemas and to allow different classes of users to interact with persistently stored collections of texts. The main problems that we observed in the approaches discussed so far are compared with the solutions provided by our approach.

1. **Limitations on Overlap**: the main drawback when using a markup language like SGML or XML to encode literary texts is the difficulty of representing multiple, concurrent hierarchies. While different approaches exists to overcome such limit, they are substantially workarounds. The XML data model is, in fact, a labeled, ordered tree structure, while a graph structure is needed to represent concurrent hierarchies. These workarounds complicate the encoding and usually they allow concurrent but not independent elements to exists. Different approaches, like the use of an ad-hoc textual model, allow the use of concurrent, independent hierarchies natively. The TOMS model, for instance, allow textual object to be in a *PAR* relation, meaning that they share a same portion of text in a non-hierarchical way. The MdF model, instead, takes a different approach: all the objects are subsets of the basic building block set, the monad set, so that the hierarchy of objects is in fact a two-level hierarchy where all objects can overlap freely, even among the same type.

   The Manuzio model solve the problem of concurrent hierarchies by the use of a graph-based data model that allows a straight representation of such structures. Textual object types are organized in a schema where each textual object has a set of other textual objects of different type as components. While the graph has to be acyclic to prevent loops in the structure, there are no other limitations on the structure, such as the number of components an object can have and so on.

2. **Limitations on Structure Definition**: XML is an ordered, labeled, tree structure. The core XML standard imposes no restrictions on the labels that appear in a given context; instead, each document may be accompanied by a document type, also called schema, describing its structure. A number of schema languages has been proposed[4].

   The first approach in the field of schema languages has been the XML DTD, a subset of SGML DTD. The use of DTDs, however, is limited:

   - their syntax is not XML;
   - they do not support namespacing;
   - they have a limited set of predefined types and no support for user-defined types;
   - attributes can not be in an exclusive or relation, nor conditional definitions can be given;

[4] A comparison of the early XML schema languages can be found in [Lee and Chu, 2000]

- they do not support any form if inheritance.

While other schema languages overcome such limits, the most important schema in computational humanities, the TEI schema, is in fact defined as a DTD, making this approach the *de facto* standard of the field. Different approaches, like XDuce or RelaxNG, could allow the specification simpler, yet more expressive schemas for literary texts.

Ah-hoc solutions like textual models, instead, have their own language to define the structure of the texts they represent. In the TOMS, for instance, such structure is defined by the declaration of relationship between objects. It is not possible, however, to define complex relationships, attribute constraints, or to limit the cardinality of repetitions. The MdF model takes a different approach, the text does not need to conform to a schema of textual object types, but structural relationships can be inspected instead from the query language dynamically.

In Manuzio the schema is defined through the use of a set of declarations given in the Manuzio language. These declarations represent a specific instance of a Manuzio model that must be valid for each document of a textual repository. By using structured data we allow the Manuzio language to have a statically checked, strong type system, where a number of programming errors can be caught before the actual execution of the program. This is of particular importance in a language with persistent capabilities where operations are, in general, time expensive. On the other hand, however, the use of a fixed, structured, schema in a textual repository limits the flexibility of the system, and assume that the structure of the text is known a priori by the encoder. Some analysis, in particular the ones related to text criticism, may find such assumption too restrictive.

3. **Limitations on Annotations**: the main limit of the markup approach when dealing with literary annotations is the impossibility of using complex, structured annotations. In XML all attributes are in fact strings, which can be interpreted as different types only if the schema language allows so. Moreover, only single elements can have attributes, so that it is difficult to annotate non-contiguous portions of text. Moreover, marking an element with an array of attributes is not natively possible, so that it is hard to model situations like a set of comments on a textual object, or an history of annotations.

   In the field of annotations even other approaches tend to be vague. The TOMS annotations seems to be not clearly defined by the model, while the MdF is explicitly generic by modeling annotations, called features, as partial functions from textual objects to values of any type.

   In Manuzio, annotations are typed and their type can be of arbitrary complexity. Since the schema language is a component of the full Manuzio pro-

gramming language it is possible to declare annotations of any type, including basic types, constructed types, or other textual object types. Moreover, for the same reason, an annotation can be a function that takes in input a textual object and, eventually, other parameters, and produces a computed results when invoked.

4. **Limitations on the Associated Query/Programming Language**: the `XML` language has a rich set of efficient query languages, like *XPath* or *XQuery*, to retrieve specific elements from a document. These languages, however, perform poorly in terms of efficiency and usability when used to query `XML` documents that represents multiple concurrent hierarchies through workarounds. Solutions have been proposed in [Iacob et al., 2004, Iacob and Dekhtyar, 2005a, Iacob and Dekhtyar, 2005b], but the resulting language, while efficient and usable, is still not optimal for literary analysis applications. Moreover, the use of such languages in complex programs is difficult because of the paradigm mismatch between the programming language and the query language.

   Textual models often employ an ad-hoc query language. The `MdF` model for instance, is paired with `MQL`, "full access language" that lets users create, delete, modify and query the objects of and `MdF` database. The query aspects of `MQL` are structural, in the sense that "the structure of the query mirrors the structure of the objects found by the query in terms of nesting, consecutiveness, and arbitrary space"[Petersen, 2004c]. On the other hand the `TOMS` system do not feature an ad-hoc query language. The documents indexing can be accessed through the functions contained in a C library. Another, more interesting, `TOMS` implementation, instead, extends a Perl interpreter to allow a direct interaction with the textual database.

   Manuzio takes a different approach. One of our goals was to minimize the paradigm mismatch between the programming language used to write the algorithms and the query language used to access the data. In the Manuzio language there is a native textual object type constructor to instance textual object types that are in effect like any other type of the language. The persistent module of Manuzio take care of translating operators on those types to calls to the textual repository's API, so that the programmer can transparently operate on textual objects just as on any other value. To simplify access to the persistently stored data, a set of query-like operators with a syntax similar to the one of `SQL` have been defined. The queries created with these operators, however, are part of the language, and so subject to typecheck as any other of the language's expressions. While, on one hand, this solution allow the programmer to stay in the comfortable programming language paradigm while writing textual analysis programs, on the other hand this approach performs an automatic mapping between the language's expressions and the textual repository primitive operations, and considerations on

efficiency and optimizations have to be made.

5. **Cooperative Annotations**: We are not aware of any general system that gives a vision of the textual data as a persistent, remote, object that can be accessed and annotated concurrently by different users with different privileges.

   In Manuzio, instead, annotations can be accessed, modified, or created on the basis of user permissions. Each annotation made on an object is stored together with a reference to the user who made it. Annotations also have an history, so that when an annotation is modified the old version is stored for comparative analysis.

In this chapter the main goal of the thesis has been presented and motivated through an analysis of existing digital text representation approaches drawbacks. Particular attention has been given to the problem of literary texts with overlapping hierarchies and to the expressiveness of annotations. The proposed solutions have can be categorized in `XML`-based solutions and non `XML`-based solutions, also called textual models. Both categories have advantages and shortcomings that have been taken into account during the design of Manuzio. In the last section a recap of the main drawbacks of the discussed encoding techniques has been proposed, and a survey on the characteristics of the current implementation of our solution has been made.

During the Ph.D. work at the base of this thesis the following goals have been achieved:

- The Manuzio model for texts has been formally defined. The model allow textual data to be structured as a directed acyclic graph where nodes represent portions of text and arcs represent a containment relation between them. A graphical representation of Manuzio models has been also defined to visually represent textual schemas.

- The Manuzio programming language was designed. With the Manuzio programming language it is possible to both define schemas and create textual repositories and connect to existing repositories to access and analyze data. While the textual schema declaration portion of the language has been specified only by its syntax and some examples, the rest of the language, a full programming language with support for transparent access to textual repositories, has been specified with a formal semantics.

- The first specifications of a system to store textual data and allow users with different permissions to query and annotate it has been given. The system feature a persistent store for textual data, an interpreter for the Manuzio language, and a graphical user interface to allow users of different levels (programmers or experts of the domain) to exploit the system's capabilities.

- A prototypal implementation of the most important parts of the Manuzio system has been developed. The textual repository has been implemented with a relational database, and an interpreter for the Manuzio programming language, written in Ruby, allows programmers to use a functional-object-oriented programming language to write programs and access transparently textual data with the full benefits of static type checking.

An interpreter for the textual schema declarations and different, more complex, user interfaces are in course of development as well as a set of standard parsing algorithms and a more efficient implementation of the language interpreter.

In the rest of the thesis the components of Manuzio, the model, the language, and the system, will be presented in turn. After that, the implemented prototype will be then discussed along with some examples.

3

The Manuzio Model

"We chose to do this work mathematically, which has the advantage of precision but is not always appreciated by readers." – Luigi Luca Cavalli-Sforza

3.1 Introduction to the model

The Manuzio textual model considers the textual information from two different points of view: as a formatted sequence of characters, as well as a composition of logical structures called *textual objects*. These objects will be defined both in terms of the text which they represent (called the *underlying text*), as well as in terms of the other textual objects which are related to them with two different aggregation mechanisms: *composition* and *repetition*. Textual objects can have also *attributes* and *methods*, and are classified through a set of types, called *textual object types*, among which a *specialization* relation is defined.

The Manuzio model takes inspiration from a number of other data models present in literature and combine some of these model's features with new ones to adapt to textual data. Our model concept of textual objects and object types is similar to that of objects and classes in object-oriented languages [Bruce, 2002]. The idea of a database dependent textual schema organized as a lattice and other features comes from the concept oriented model discussed in [Savinov, 2008]. While the Manuzio model is not concept oriented, it shares some of these model's terminology and characteristics.

In the following sections an introduction to the model, with the aid of a graphical syntax, and its formal definition are given. Here Manuzio is presented as an abstract model, so an implementation will not be specified until the prototype presented later in Chapter 6.

3.2 Model Definition

Text

The basic concept of the Manuzio model is the concept of text. Since we are presenting Manuzio as an abstract model the text concept will be abstracted, without loss of generality, to that of a finite, ordered sequence of characters expressed in any encoding system. In a real implementation such text can be represented in a multitude of ways like relational databases data fields, xml documents, and so on. We will call this entity the *full text*, and it will represent the whole text of a Manuzio textual model instance.

Definition 1 *The* full text *is a sequence of Unicode characters that represents all the text described by a specific Manuzio* textual model*.*

The full text entity has some associated operations needed to access contiguous portions of the text using some indices. Since we are considering such text as an ordered sequence of characters we can isolate a portion of such sequence by specifying a pair of integers in the form (index, offset). Such pair is called a *range*. We can then enumerate the elements of the sequence from zero to n, where n is the cardinality of the textual sequence, and use the range to obtain a contiguous *slice* of the text.

Definition 2 *If t is a textual sequence in the form $t = [c_1, c_2, \ldots, c_n]$ and r is a range in the form $(index, offset)$ with:*

$$0 \leq index \leq n$$

$$index < offset \leq 0$$

then the expression

$$t :: r$$

is called the slice of t given by r and is a new sequence of characters with the same encoding as t composed by the characters of t in the range r.

As we will see in the rest of the chapter the slice operation will be of central importance in the model. Programming such operation, while conceptually simple, can be an implementation challenge when the text to be modeled is large. A panoramic on possible approaches is given in Chapter 6.

Textual Objects

A *textual object* is an abstract representation of a portion of the full text together with structural and behavioral aspects. Textual objects have been inspired by the content objects expressed in [Bruce, 2002], from concepts of concept-oriented data modeling [Savinov, 2005a], and from objects of modern object-oriented programming languages.

Definition 3 *A* textual object *is a software entity with a state and a behavior. The state defines the precise portion of the full text represented by the object, called the* underlying text*, and a set of* properties*, which are either component textual objects or attributes that can assume values of arbitrary complexity. The behavior is constituted by a collection of local procedures, eventually with parameters, called* methods*, which define computed properties or perform operations on the object.*

Usually a textual object is the instance of a logically identifiable concept in the text, like a paragraph, a chapter, a word, and so on. Textual objects can be in relation with each other. The main kind of relationship we are interested in exploiting is the *component relation* between textual objects.

Definition 4 *A textual object TO_1 is a* component *of a textual object TO_2 if and only if the underlying text of TO_1 is a subtext of the underlying text of TO_2.*

It is important to note that the concept of *subtext* is not equivalent to the concept of substring. According to [Kelley, 1995]:

> *A substring (or factor) of a string $t = [c_1 \dots c_n]$ is a string $t' = [c_{1+i} \dots c_{m+i}]$ where $0 \leq i$ and $m + i \leq n$.*

According to the definition a substring is a contiguous portion of the original string. The concept of subtext, instead, is broader, as a subtext can comprise non-contiguous parts of the original string.

> *A subtext of a string $t = [c_1 \dots c_n]$ is a string t' in the form $t' = [c_{1+i_1} \dots c_{m_1+i_1}] \cup \dots \cup [c_{1+i_2} \dots c_{m_2+i_2}]^{1 \leq j \leq k}$ with, for each $j \in 1 \dots k$, $0 \leq i_j$, $m_j + i_j \leq n$, and $[c_{1+i_1} \dots c_{m_1+i_1}] \cap \dots \cap [c_{1+i_2} \dots c_{m_2+i_2}] = \emptyset$.*

This is an essential aspect of the model and has an important consequence on textual objects: a textual object can consist of a repetition of components that does not need to be contiguous.

Definition 5 *A* repeated textual object *is either a special object, called the empty textual object, or it is a homogeneous sequence of textual objects, and its underlying text is the composition of the underlying text of its components.*

Another important concept is the concept of *contiguous* textual objects. Two textual objects are contiguous if their underlying text seen as a portion of the full text is contiguous.

Since sequences of textual objects, like the sequence of words of a sentence, or the sequence of acts of a poem, will be an entity of central importance in our model, we use the concept of repeated textual objects to abstract their characteristics. Differently from a classical sequence a repeated textual object elements cannot contains duplicates. The ordering of the elements is induced by the position in the text of their underlying text, so that it cannot be changed. Moreover, in our model, repetitions can have gaps, so that it is not required that their elements are contiguous[1]. For instance we can consider the first three words of a sonnet as a repeated textual object, all the lines of a poem another one, all the first lines of Shakespeare's plays another, and so on.

Textual Object Types

Each textual object is an instance of its textual object type. These concepts have numerous synonymous used in different models. Textual objects could also be called items, entities, values, while textual object types are also called classes, concepts, or domains. In the rest of the thesis the "textual object type" will be abbreviated with "type" when such abbreviation will not raise ambiguities in the text.

Definition 6 *A* textual object type *specifies the type name, the names and types of the properties, as well as the names and the parameters of the methods together with their types of the textual objects that are instances of that type. The type of a component is always another textual object type, while the type of an attribute is a data type, like integer, string, boolean, a record type, etc. The parameters and result types of a method can be of any type.*

A textual object type T is characterized by an *intent* I and an extent E. The intent is a set of variables associated with the textual object type, also called attributes, properties, or features in other models. The extent of a textual object type is a set of textual objects that are instances of that type. The intent of a textual object is partitioned in *components* and *attributes*. While attributes are values of any type that represent extra-textual information, components are used to store information about textual object type's relationships. We can denote a textual object type with $T =< C, A, E >$, where C is the set of the type's components, A are the attributes, and E its extent.

Repeated textual objects also have types, and such types are more than simple type constructors like, for instance, the sequence type constructor of programming languages. A repeated textual object type can include other useful information about the collection of textual objects they represent.

[1]Here the intuitive notion of contiguous objects according to their underlying text is used.

Definition 7 *A* repeated textual object type *R is a type which instances are repeated textual objects. Each repeated textual object type is an aggregation of a type T, the textual object type of its* elements*, and of a collection of variables of heterogeneous type called its* attributes*.*

Differently from textual object types, repeated textual object types does not have components, but elements, while both can have attributes. This distinction is important because it gives a wide flexibility to the text annotation process. A repeated textual object composed by the three words, for instance, can be annotated with a comment. Such comment will be tied to the three words as a set, but not to them individually. A repeated textual object type, also called *repetition* for brevity, can be denoted with $R = [T]$, where T is the type of the repetition's elements.

To simplify the model we can define a special bound between textual objects of type T and repeated textual objects composed by elements of type T.

Definition 8 *A textual object type T has an associated plural form, usually named with the plural form of its type name. The repeated textual object S which elements are of type T and name is the plural form of T is called the* plural type *of T. Conversely T is called the* singular type *of S.*

For example a `Poem` can have a component with name `title` and type `Sentence`, as well as a component with type `lines` and type `Lines`, which is the plural form of the type `Line`. This means that the type of the lines of a poem is a repeated textual object type with elements of type `Line`.

In our model each textual object type can have only one plural form. For instance, if T is a type that represent a word, only one repeated textual object type S with T as elements can exists. We feel that the trade off of having such a simple and powerful conversion capabilities between singular and plural types justify this slightly restriction of the type system.

The Component Relation

The component relation among textual objects is naturally extended to their types.

Definition 9 *A type T_1 is a* direct component *of a type T_2 if there is a component in T_2 which is of type T_1 or of its plural form T_1s. A type T_1 is a* component *of a type T_2' if it is a direct component of T_2' or if it is a component of some of its direct components. Given two types T and S if T is a component of S then S is also called a* parent *of T. Moreover, given two types T and S, either S is a component of T or vice-versa, but not both.*

The component relation is asymmetric and transitive. When a type T is a direct or indirect component of another type S we can also say that there is a *link*

between such types. When we say there is a link between T and S we mean that S is a component (either direct or indirect) of T, and not vice-versa.

The component relation can be modeled as a graph where nodes are textual object types and arcs are links. Since the relation is asymmetric the resulting graph is acyclic. It is also directed and we choose to direct the arcs so that if there is a link between T and S the arc goes from T to S. Note that, since repeated textual objects can be also seen as textual objects a component relation can exists between both a type and another type and between a type and a repetition. In the first case we say that a component relation is one-to-one, while in the latter is one-to-many.

Textual Objects Semantics

The semantics of a textual object type instance, or textual object, is similar to the semantics of objects of an object oriented language. A textual object is characterized by its type and by the values of its components and attributes.

An attribute's value is an instance of that attribute's type. For instance an attribute could be of type *String* and its value could be "Shakespeare" to denote the authorship of a poem as extra-textual information. There are no restrictions to the attribute types, so that arbitrarily complex values can be stored, like records, sequences or even references to other textual objects. For a more in-depth discussion on available types and type constructors see the discussion on types in Section 4. On the other hand, components are just textual objects. They can be either textual objects or repeated textual objects of the type specified by the component definition.

Objects that have no components and are also called *primitive* objects. Primitive objects, unlike other textual objects, are atomic and can define their own semantics. Each primitive object has a special attribute, called *text*, which value is a set of ranges. Each range identify a unique portion, or slice, of the full text.

Primitive objects play the role of basic building blocks for other objects. Since in the Manuzio model objects are composed by other objects to retrieve some of their characteristics, where their underlying text is the most important one, we must proceed in a recursive way. Primitive objects forms the base of such recursion. Since a Manuzio schema is always structured as a lattice (see later in Section 3.2) such recursion is guaranteed to end. We say that textual objects cannot, in general, define their semantics on their own. Their semantics is spread over the schema. The underlying text of non-primitive objects is built by aggregation:

Definition 10 *The underlying text of a non-primitive textual object is the portion of the full text identified by the range set obtained by merging the ranges of the underlying text of its components. The underlying text of a primitive object is the portion of the full text identified by its range set.*

Two ranges can be merged as the classical mathematical ranges. Note that, since component textual objects do not need to be contiguous and primitive objects

underlying text can have gaps, this union can result in a set of subtexts of the full text rather then on an atomic substring.

Subtyping and Inheritance

As in other traditional object-oriented models, in Manuzio a subtyping relation can be defined among textual object types through which we can model textual objects at different levels of detail. This feature adds to the model the ability to make incremental changes to textual objects. A subtype may be defined from a base textual object type by adding or modifying components and attributes. We will see later that restrictions on the modification of components and attributes types are necessary in order to preserve type safety.

For instance, if a type `Work` has components `title` and `sentences`, and attributes `author` and `year`, we could define the type `Poem` as specialization of `Work`. `Poem`, in addition to *inheriting* properties and methods from `Work`, could have a new component, `lines`, and a new attribute, `meter`.

Definition 11 *A type A is subtype of a type B if it is defined as such; in this case T inherits all the properties and the behavior of S. T can also have new properties and methods, and can redefine the type of its components with a more specialized object type. In this case a value of type T can be used in any context where a value of type S is expected.*

We write the subtype relation as $T <: S$. The presence of the subtyping (or specialization) relation between two textual object types T (the *subtype*) and S (the *supertype*) has the effect that every instance of the subtype is also an instance of the supertype. That is, a value of type T can masquerade as an element of type S in all contexts where $T <: S$.

For example, every poem can be treated both as a generic work (for instance by asking for its author), and as an object with a component `lines` (for instance to count them).

The subtype relation among textual objects types is independent from the component relation, expect from the fact that it applies to the same set of types. For completeness, every type which is not defined as subtype of another is implicitly subtype of an abstract type `TObject`, which has no components nor attributes, and has only a set of basic methods common to all textual objects, such as text extraction, equality test, and so on.

A more complete discussion about how inheritance is implemented will be given in Section 4 with the Manuzio language implementation principles.

Textual Schemas

A textual schema is a set of textual object types and links so that the resulting graph is a bounded lattice.

Definition 12 *A* well-formed textual schema *of a certain full text is a set of textual object types which forms a bounded partial order set with respect to the component relation and for which: a) there exists a minimal, undecomposable type, (the* `Unit` *type); b) there exists a maximal type, called by convention* `Collection`*, which has a single instance,* `collection`*, the textual object whose underlying text is the full text and the components are the top-level components of the model.*

A textual object type can be labeled as *Unit* when it has no components. A *Unit* type represent an atomic textual information that cannot be decomposed in "smaller" logical units. The specific definition of such type is left to the text encoder, so that, for certain schemas, the unit type can be the word, for others the syllable, for others again a whole sentence, and so on. It is important to note that such property of being undecomposable is valid only at the logical level where Manuzio models the text. It is always possible, of course, to decompose any object underlying text into characters by applying string operations, but since our model aims to give an high level view of the text the unit type should be chosen, so that no such "low level" operations are needed to accomplish the textual analysis required by a particular schema.

The *Collection* type, also called *Total*, is a maximal type which always has a single instance, called *collection*, which underlying text is the whole full text. The *Collection* type does not have any parent and is a parent for all the other textual object types of the schema.

> *In mathematics a* lattice *is a partially ordered set (also called a poset) in which any two elements have a unique supremum (the elements' least upper bound; called their join) and an infimum (greatest lower bound; called their meet). A bounded lattice has a greatest and a least element, also called top and bottom.*

A textual schema is a lattice where the elements of the set are the textual object types of the schema partially ordered by the component relation, the greatest element is the *Collection* type, and the least element is the *Unit* type. A *path* in the schema consists in a sequence of types, starting from a source textual object type and ending in another type that is one of its direct or indirect components. Since the Manuzio data model is a graph there can be more than one path between two textual object types if their component relation is indirect. A path can be denoted with the following syntax:

$$T \to c_1 \to \cdots \to c_n \to S \tag{3.1}$$

where T is the source type and S is the target type. The number n is called the *rank* of the path. A path of rank 0 is a path connecting two types in a direct component relation. The set of different paths connection a textual object type to the *Unit*

type are called the *canonical path set* of a type and will be of particular importance to retrieve the underlying text of a textual object.

In Manuzio each textual database has a specific schema that is defined at design time and does not change during the database lifetime. Such schema describe the textual object types, their name, their attributes and their components, of all the logical concepts of interest in the text such as words, paragraphs, chapters and so on. Moreover, the *Unit* type is also database dependent, so that it can be different from schema to schema to accommodate different types of analysis needs.

Graphical Notation

To represent our model instances we developed a simple graphical notation resembling the object oriented languages `UML` one. In our notation textual object types interfaces are represented as boxes split in two parts. The upper part contains the name of the type, while the lower one, if present, contains the name and the types of its attributes and methods. Each attribute or method is denoted by its name followed by a colon and its type. Differently from `UML`, where classes and objects can have a wider range of relations with each other, in our model the only relations allowed between textual object types are the component and the inherited relation.

The component relation is defined by the name and type of the components of a textual object type, so in our graphical notation a component is draw as a directed arc connecting a type interface with the interface of that component type. An arc is labeled with the component name and with the cardinality of the relation. One-to-one relations are represented with a single-pointed arrow arc, with the arrow directed toward the component, while one-to-many relations are represented by similar but double-pointed arrow arcs.

In Figure 3.1 the graphical notation of a simple schema about poems is shown. The `Collection` type has a text annotation to store its title and has a components called `poems` which is a repetition of objects of type `Poem`. The `Poem` type is more complex and has three different components along with a richer set of attributes of different types. The *Unit* type of this schema is the `Word` type, since it has no components.

For what concerns the subtype relation, a subtype is graphically connected to its supertype through an arc with a hollow arrowhead pointing to the subtype. The textual object subtype interface represents only the new attributes and only arcs concerning new components are drawn.
In Figure 3.2, both `Novel` and `Poem` are subtypes of `Work` so that they inherit the components `title` and `sentences`, as well as the attributes `year` and `author`. Moreover, the `Novel` type has the new attribute `subject`, while the `Poem` type has the new attribute `meter` and a component `lines` to model lines of a poem. As a convention we draw the textual objects type interfaces from the top to the bottom, so that the collection type will be always at the top of the graph and the unit type at the bottom. Moreover, if a type T is component of another type S then T will

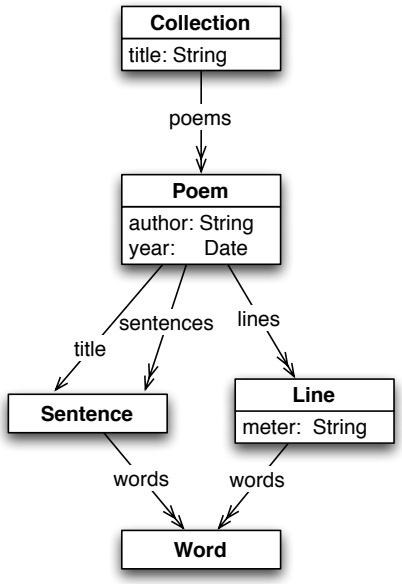

Figure 3.1: Graphical representation of a poem related schema.

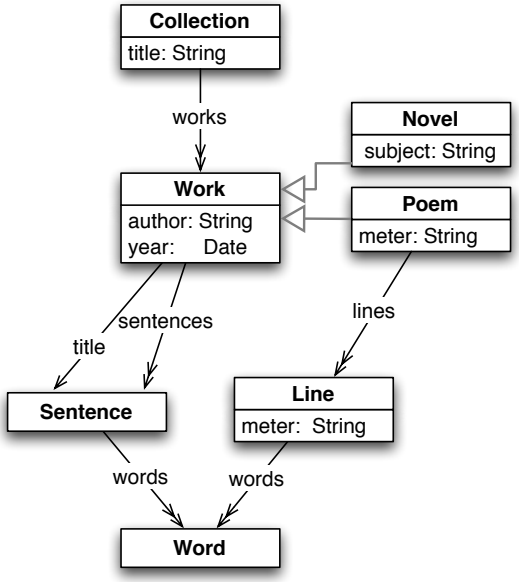

Figure 3.2: Graphical representation of a schema about poems and novels.

be drawn below S. It is always possible to draw the schema in such way since the model is a lattice. By using these conventions the resulting graphical notation is clean and easy to read.

3.3 Discussion and Comparison with Other Models

Our model have some similarities with the classical object-oriented models of programming languages. In fact, if we consider textual objects as composed only by attributes and methods, we can view them just as another kind of objects. The novelty of our approach is given by the specific composition mechanism of our textual objects which connects them in a structured hierarchy with characteristics specific to the Manuzio domain of application. The nature of this specific domain allows the construction of a bounded partial order set of textual object types so that interesting, powerful, easy to use operators can be introduced in the associated language.

While other object models like the one of object oriented languages are more powerful then the Manuzio one and permits to model complex behaviors like multiple inheritance, bounded inheritance, more complex ways of subtyping, and so on, we feel that the Manuzio object model is adequate to be used in computational linguistic applications.

Differently from the `XML` (or `SGML`) based solutions, Manuzio is a model with an associated ad-hoc programming and query language. Since this language is Turing complete programs and queries of arbitrary complexity can be written over Manuzio databases with just one, integrated language. Classical `XML` applications must overcome the problem of overlapping hierarchies and often the workarounds used to overcome this problem makes the use of query languages like `XQuery` or `XPath` difficult. While one could argue that `XML` is a wide spread standard and the use of different models, like Manuzio, could hinder research projects, we feel that by including algorithms to export views of a Manuzio database, or query results, in `XML` largely overcome this problem.

Other textual models like `TOMS` of `MdF` have been of great inspiration to our work. Differently from `TOMS`, where the minimal object is always considered to be the word, in Manuzio the selection of such object can vary between different schemas, increasing the overall flexibility of the model. Moreover it is possible, through the Manuzio language, to define constraints on the schema, for instance on the cardinality of elements in a repetition. In our opinion, moreover, the choice of constraining schemas graph to a lattice and to avoid disjoint union of elements helps the programmers to express programs and queries easily. We felt that the root-level parallel structures and the "choice" construct of `TOMS`, while adding flexibility to the model, could be an unneeded complication for the users.

The `MdF` model have a strong and sound mathematical model based on the con-

cept of monad. The main difference with our model is that, in MdF, objects are composed of monads, and not by other objects. For this reason MdF lacks the concept of schema, and it is not possible to constraint at model level relationship between objects. On the other side such freedom makes this model more suitable for texts that lacks a precise schema, or for application where an heavy amount of text criticism is needed. The MdF model is also paired with a powerful query language called QL and subsequently extended by MQL[Petersen, 2004a], that allows users to perform structural searches in very intuitive and easy way. While very different, in goals and design, MQL inspired the presence of ad-hoc constructs to deal with textual object hierarchies in the Manuzio language.

While both the XML-based models and TOMS can handle annotations only in string format, the MdF model can handle annotations, or features, of any kind, but their specific implementation is left open. In most application of such model annotation are just strings or string-encoded values that the user have to interpret when performing queries. Manuzio benefit from the presence of a Turing complete, type safe, strongly typed language that ensure a full support to structured annotations of arbitrary complexity by the use of a rich set of types and types constructors that are natively present in the Manuzio programming/query language. An interesting aspect of the Manuzio model is, in our opinion, the ability to reference repeated textual objects, a kind of sequence that occurs frequently in our domain of application, in a native, easy way. With the introduction of repetitions the Manuzio language will be able to treat repeated textual objects as single ones, in a simple and uniform way.

4

The Manuzio Language

"Languages are not about what they make possible, but about what they make beautiful" – Scott Davis

4.1 Motivations of the Manuzio Language

In this section we present the Manuzio programming language, whose main goals are:

- to provide a syntax to define textual models;
- to be a query language for textual databases based on the Manuzio model;
- to be a Turing complete programming language that natively integrates persistence and query capabilities to interact with textual databases.

One of the first considerations to be taken when dealing with a new language is: was it really needed? The field of programming languages is overloaded by a huge number of new languages that dies within one year. Indeed, other approaches could have been followed in the development of the Manuzio language.

- *Implementing Manuzio as a library.* Since the model relies on objects that are similar to object-oriented language objects one of the most straightforward ways to design our language could have been to choose a suitable, existing, object oriented language and develop a library to add the textual object concept to it. This solution grants good performances and, most of all, by choosing a wide spread language we can ensure a good base of users that are familiar with the syntax and the construct of such language. At the same time, however, we felt that choosing this solution was not interesting enough for a research work. It is for sure the best way of getting something done in a short time, but it is not suitable to research new programming constructs useful to the specific domain of application, since we must rely on the constructs of the existing language.

- *Extending an existing programming language.* To overcome some of the previously discussed problem we could have designed the Manuzio language as an extension of another language. By extending we mean to add capabilities directly to the language compiler or interpreter, in our case to support textual objects. While this solution permits to add new constructs and behaviors that are not present in the original language of choice, we felt that our results could have been biased by the language choice. During the development, for instance, different solutions have been carried out for a number of features such as making Manuzio a class-based or object-based language, or choosing between name or structural equality. By using an existing language such choices must be made in advance, while we needed a more free environment to test and play with. Indeed, with the results presented in the last chapter of this work, we can define a suitable language to implement Manuzio with, and choose this solution for a more advanced, efficient compiler or interpreter.

Our choice to implement a completely new language grants an unbiased, free environment to experiment new features such as partial persistence and textual objects support. We are aware that the resulting language, as discussed in Section 6, will not be adequately efficient nor it will have wide acceptance to be used for more than prototypal systems. However the obtained results will be of great use to design and develop a real-life, efficient, language, or a language extension, that implements the Manuzio model.

In the rest of the section a quick overview of the main Manuzio language characteristics is given. We start with an introduction to functional programming and functional programming languages and we then move on to discuss the most important aspects of our language by an informal description and examples. The goal of this chapter is to give an introduction to the language and its features. The formal language specification will be the argument of the next chapter.

4.2 A Brief Introduction to Functional Languages

The topic of functional programming languages is broad and a complete discussion is beyond the scope of this thesis. Only the main features of such languages that are useful to understand the Manuzio language concepts will be presented here as an overview. Further information on functional programming languages, types, and other related concepts can be found in [Scott, 1999].

Functional programming has its roots in *lambda calculus*, a formal system for function definition, application, and recursion introduced by Church in 1932 as part of an investigation in the foundation of mathematics [Selinger, 2001]. In functional programming computation is treated as the evaluation of mathematical functions, avoiding states and mutable data. A *function* is a mapping between elements of two

sets:

$$f : A \to B \tag{4.1}$$

this notation indicated that the function f takes in input a value of the set A, called its *domain*, and returns an element of type B, called *codomain*. With this approach the focus of programming is given on what is to be computed and not how we compute it. The application of a function to an argument is usually denoted by juxtaposition, as in $fact(n)$ that indicates the application of the function $fact$ to the argument n.

The approach used to evaluate programs is radically different from the one of imperative languages. In a functional languages the *expressions* of a program gets evaluated and the results are displayed to the user. The process of evaluation is a simplification process also called *reduction*. Starting from a language expression the goal of reduction is to obtain a value, or an *normal form* associated to the expression. It is also said that the *meaning* of an expression is its value.

Reduction steps are usually denoted as follows:

$$e \to e' \tag{4.2}$$

meaning that there is a reduction step that transforms the expression e in an expression e' with the same meaning. For instance to evaluate $square6$ the evaluator replaces the function application of $square\ x = x * x$ with the function body and the parameter x with the actual value 6, obtaining the new expression $6 * 6$.

Example 2 *Consider the expression:*

$$(1 + 1) * (2/2) \tag{4.3}$$

Assuming that the reduction takes place left to right we can simplify this expression as follows:

$$(1 + 1) * (2/2) \to (1 + 1) * 1 \to 2 * 1 \to 2 \tag{4.4}$$

We can say that the value, or meaning, of the initial expression is 2.

In the previous example we assumed the reduction to be executed left to right, but other approached could have been possible, like right to left. In both cases the value of the expression is the same. This is a property of functional programs called *unicity of normal form*: the value of an expression is independent from the order of reduction.

All functional languages allow functions to be passed as arguments to other functions, or returned as results. Functions that take functional arguments are called *higher-order* functions.

Programs in functional languages are in general shorter, easier to understand, debug, and maintain than imperative ones. The main domains of application of functional languages until now have been in the field of artificial intelligence, text processing, graphical interfaces, natural language processing, telephony, music composition, symbolic mathematical systems, theorem provers of proof assistants [Fernandez, 2004]. The ancestor of functional languages can be considerer `LISP` (LISt Processing) a language born to process lists of numbers and characters that has been introduced in the 50's [Steele, 1990]. In `LISP` both programs and data are represented as lists, so it is easy to define higher order functions in `LISP`. Functional languages evolved from the untyped, dynamically scoped, `LISP` to more complex languages with static scoping and a strong type system like `ML` and Haskell, etc.

Functional programming languages discussed so far are also called a *purely functional*, or *pure*, language. In other words, the functional programming paradigm avoids the use of *state* and *mutable* data, so that evaluation of an expression is independent of its context. Haskell, for instance, is a popular academic pure functional language. On the other hand languages like `ML` are functional languages that have, often for efficiency reasons, some imperative features. Purely functional programming languages have several useful properties, many of which can be used in code optimization:

- If an expression result is not used it can be removed without affecting the rest of the program.
- Since the result of a function call is constant and does not produce any side effect, its value can be memoized so that, if the function is called again with the same parameters, there is no need to execute again the function body.
- The evaluation of a purely functional language's expressions is thread-safe. Since there is no side effect and no data dependency between the body of to different functions, they can be executed in parallel without issues.
- The evaluation strategy of purely functional languages does not influence the results of their computation.

Another distinctive feature of functional languages is the use of recursion: in the definition of a function f it is possible to use the function f itself. Recursion is the functional counterpart of iteration, one of the main control structures of imperative languages.

In a functional programming language the value of a function is represented by a *closure*: a record that contains the code of the function body and the values of the necessary non-local variables. The concept of non-local variables, also called the *free variables* of a function, is similar to its mathematical counterpart: in programming a free variable is a variable referred to in a function that is not local nor an argument for that function. Such concept have a meaning only in a *statically scoped*

environment, where the values of free variables are always the values they had at the time of the function definition.

Some functional languages implements *lazy evaluation*, also called call-by-need. In lazy evaluation each variable is not a value but a *thunk*: a function that is invoked on demand to compute the variable value. Each thunk is also equipped with a cache: when the thunk is called the first time it computes its value and memoize it in the cache. The next time that such value is needed the thunk does not get executed but the cached value is returned instead. With lazy evaluation values get computed only one time and only when they are really needed, so that unused values are never calculated.

In most functional languages values are categorized into types according to the operations that can be performed on them. If a functional language is typed every expression has an associated value of a certain type. Types are generally used as a complement to programs: they are concise descriptions of programs which goal is to detect errors before the program gets executed and to allow the compiler to perform code optimizations. A typed programming language comes with a set of predefined *basic types* such as integers, reals, characters, booleans, and so on, and with a set of *type constructors* to create complex, structured types like lists, tuples, functions, and so on.

The process of checking a program for type errors is called *type checking* and may occur at compile time, under the name of *static type checking*, or at execution time, under the name of *dynamic type checking*. In a *strongly typed* language expressions that cannot be typed are considered erroneous and are rejected by the compiler or interpreter prior to evaluation. Type systems can be classified as *monomorphic* or *polymorphic* depending on the number of types that an expression can have. In a monomorphic language each expression has exactly one type, while in polymorphic ones some expressions can have more than one type. Such behavior can be obtained througin several ways:

- Generic types: a type variable can be defined as generic and be instantiated with an actual type at a later time in the program.
- Overloading: several functions with the same name can work on different types. The choice of the specific function to use can be made statically by examining the parameters type.
- Subsumption: if the type system supports subtypes than the subtyping relation can allow functions to use different, compatible, types.

Functional languages usually achieve polymorphism through generic types.

Example 3 *Consider the identity function* $f(x) = x$. *The type of this function is* $X \to X$, *where* X *is a generic type. Such type can then be instantiated to, for instance,* $Integer$ *to obtain the integer identity function of type* $Integer \to Integer$ *and allow expressions such as* $f(3)$.

Most modern functional programming languages offer a *type inference* mechanism. In a language with type inference the programmer does not need to explicitly write the type of expressions. The type inference algorithm can *infer* their type based on the type of their primitive values and operations. Types can still be declared so that the programmer can force type control.

Example 4 *Assuming that:*

$$let\ x = (5 * 5) \tag{4.5}$$

we can infer the type of x *by knowing that* 5 *is a constant of type* $Integer$ *and that* $*$ *is an operation of type* $(Integer, Integer) \rightarrow Integer$*. The type checker first reduces the* $*$ *operation and obtain an* $Integer$ *value. Then create a new constant* x *and assign the* $Integer$ *type to it. If, for instance, the programmer would have declared:*

$$let\ x : Boolean = (5 * 5) \tag{4.6}$$

this would have lead to a type clash and to a rejection of the expression from the type checker.

An important concept when discussing a type system is the concept of type equivalence. In a *structural* type system type equivalence is determined by the type structure. Different type systems check such properties basing their decision on explicit declaration (*name* equivalence) or by checking dynamically if parts of the type structure accessed at run time are compatible (*duck* typing). In structural typing two types are considered to have the same type if they present the same structure. The exact definition of structure depends on the language semantics. The main issue of structural equality is the inability of distinguish between types that the programmer may think are different, but which happens by coincidence to have the same internal type structure.

This simple introduction to functional languages should be enough to understand the main concepts of the Manuzio programming language presented in the following section.

4.3 Overview of the Manuzio Language

The Manuzio language is a programming language built to be highly dynamic and extensible. For this reason, differently from the majority of other programming languages, Manuzio does not have a fixed set of types, operators, expressions, and so on. The Manuzio language can be seen, instead, as an empty core language that defines basic behaviors, together with a set of additional components, called *bundles*, that are used to import new functionalities into the language. The concept

of extensibility is discussed in depth later in Section 4.3.1. In this section, however, the core and the most important bundles will be presented together to give an overview of the whole language. An in-depth discussion about the single bundles and their structure can be found in Chapter 5.

The Manuzio language has the following features:

1. It is an expression language: every construct returns a value.
2. It is an interactive language: the system prompts the user for input and returns the results of computation.
3. It is higher order: functions are denotable values of the language, so that they can be passed as parameters, used as record fields values, and so on.
4. Every denotable value of the language has a type:
   - a type is a set of values sharing common characteristics, equipped with primitive operators which can take these values as parameters;
   - the predefined types of the language are booleans, integers, strings, ranges, and so on with their classical operators, together with the command type used to denote non-functional constructs and the null type, a singleton set with the element nil only, used to type undefined values. Strings, in particular, are an important basic type in a textual analysis program, and their type is equipped with a rich set of operators such as pattern matching;
   - type constructors are available to define new types, like sequences, tuples, objects, and so on. Each type constructor can accept as parameters both primitive types and other constructed types;
   - the type system supports subtyping, so that subsumption can be used to achieve a form of polymorphism.
5. Every expression has a type, that is the type of the value it returns. Types of expressions can be statically determined, so that semantic errors in programs can be detected by a static type check. Static type check brings a considerable benefit in testing and debugging programs and allows type information to be discarded at runtime. Moreover, a static type checker allows the language to give the correct meaning to overloaded operators before the actual program execution.
6. A special kind of objects, called textual objects, are equipped with persistent capabilities so that, by issuing a special command to the language, a particular set of predefined textual object types and instances gets loaded. Each of those sets of types and instances are the digital representation of a collection of texts organized in a Manuzio model, and can be used in textual analysis programs.

The main goal in the Manuzio design has been to make it a language that can be approached in a dual way: a programmer can use it to write programs of arbitrary complexity, while a humanities researcher can, even without a full programmer background, use it in a way similar to that of a query language to interactively interrogate a textual database.

4.3.1 Extensibility

In the development of Manuzio we choose to use a newly created language to test our model in a non constrained environment. To achieve such goal in full we decided to design our language following some ideas presented in [Albano and Procopio, 1998]. Manuzio can be considered "dynamic", meaning that it is easy to extend with new types, expressions, or operators. In the same way it is also easy to remove features without affecting other functionalities.

To achieve such goal Manuzio has been designed initially as an empty language with basic behaviors but no types, values, or operators. This empty language describes only the general rules of Manuzio: how the type checker works, how the environments are structured, and so on. Later, it has been extended with new elements to reach a full featured functional object-oriented language with specific constructs and operator for textual analysis. Each element of Manuzio can be identified as a logical portion of a programming language such as a specific construct (or family of constructs) or a data type. Examples of elements are: integers and their associated arithmetic operators, records and their operators, variable declarations, the if-then-else construct, and so on. The empty language, called $\mu Manuzio$, has no expressive power and it is not possible to write anything more than the empty program with it.

Additional elements are introduced as part of *bundles*. A bundle is an interpreter extension that can add new types, values, constants, or expressions to the language. Each bundle specifies its specific static and dynamic semantic rules, so that every bundle represents a self-contained portion of the final language. Some bundles can depend on others. It would be impossible, for instance, to define textual objects without knowing about objects, or regular expressions without strings. When a bundle B depends on a bundle B' we say that the bundle B is *required* by B'. If B is not required by any other bundle of the language it is possible to *remove* B from the language. Removing a bundle creates a new interpreter without that bundle functionalities. In the current implementation bundles must be disjoint, so that it is not possible, for two different bundles, to define the same operation twice.

The Manuzio interpreter is designed in an object-oriented way, where each bundle of the language is an object of an object-oriented language. For this reason it will be convenient, but not required, to implement it in an object-oriented language of choice. In the next section the Manuzio language's formal specification will be given following the principles presented here, so that the empty language rules will be defined first and then each bundle will be presented following a rough general-to-

specific order. The dependency relation between bundles will be explicitly given and represented by a graph.

The adoption of this design has the benefit of making the language highly dynamic. By taking into consideration only the bundles dependency relationship we can add or remove features from the language without issues. In the course of the development, for instance, we gave Manuzio the concept of memory and variables, making it an impure functional language, to allow the users to modify annotations on textual objects. Later on we removed such capability by just disabling the correspondent bundle without the need of touching the interpreter code. Disabled bundles can be reintroduced in the language at anytime. The only precaution that has to be taken manually, for now, is to avoid name clashes of types and values between different bundles. With this powerful implementation of our interpreter it is easy to test new features of the language, restrict the language capabilities, or modify the existing ones without breaking other bundles behaviors. We found this design a well-suited platform for our research.

4.3.2 Type System

Manuzio is strongly typed with a static type system. Each expression of the Manuzio language must pass the type checker before execution. Expressions that cannot be typed are not executed to prevent runtime errors. The type system of Manuzio can infer types, so that it is not necessary to always declare them. Type equivalence is calculated by structure, so that if two types have the same structure they are also considered the same type.

While according to some opinions structural type equivalence can lead to ambiguity in programs [Pierce, 2002], our choice of structural equivalence is motivated by two reasons. Firstly, it copes well with the concept of modular implementation discussed later, in Section 6. Secondly, the main focus of Manuzio is on textual objects. Such objects are less general than standard object-oriented languages objects, since they all represent the concept of "a portion of text with some logical meaning". With this assumption it feels perfectly reasonable that two different textual objects are treated as the same if they have exactly the same components and attributes. For instance, if a line and a sentence are structurally equivalent, it could be reasonable in some contexts to pass a line to a function that works on sentences and vice-versa. This behavior helps to achieve a certain degree of freedom when writing programs in a way similar to dynamic typing while still preserving type safety.

The Manuzio type system also supports subtypes, and the exact subtyping rules will be specified with each new type introduced by bundles. Subsumption between subtypes is supported, so that when a type T is subtype of another type S then a value of type T can be used wherever a value of type S is expected.

While by exploiting structural equivalence we can apply code written for a certain type to another, structurally equivalent, type, with *polymorphism* we can apply the same chunk of code to different types. Manuzio implements polymorphism through

parametric polymorphism: it is possible to declare generic functions that takes type parameters and returns actual functions as result. A form of ad-hoc polymorphism is implemented in built-in operators so that some of them can be applied to different types. For instance, as we will see in depth in the next section, the *plus* (+) operator works for integer and real numbers as well as strings. Another case is the *dot* (.) operator is used to access record fields, but works also on objects and textual objects. Since the Manuzio type system supports subtypes subsumption can also be used as a mean of achieving polymorphism.

The predefined types of the language are *integer*, *boolean*, *real*, equipped with a minimal set of operators. *Strings*, instead, are equipped with a richer set of operators to allow comparison, computation of distances (like Levenshtein distance), and pattern matching with regular expressions. Other primitive types of the language includes *command* and *null*. Both are singletons types with the value, respectively, of *nop* and *nil*. The former is used to denote the value of expressions that have side-effect. The latter is used, instead, to denote the value of unassigned memory locations, unknown values, and so on. Also ranges and regular expressions types are treated as primitive in Manuzio due to their importance when dealing with textual data.

4.3.3 Declarations

In the Manuzio language two concepts of declarations coexists. On one hand we have the classical concept of variable declaration, an expression that can be used to add a new bind between an identifier and a type or a value to the language's environments. On the other hand a special block, called the *textual schema declaration block*, can be passed to a Manuzio program to declare the names and types of a schema's textual objects.

When a textual schema declaration block is encountered, a new schema with the supplied name is created in the textual repository. The internal persistent storage is then initialized with the specified textual object types so that it is ready to be instanced with textual objects instances. During the parsing process of such block a parsing graph is created, so that the constraints specified in chapter 3 can be checked. A schema that does not satisfy the lattice-like structure required, for instance, would be rejected at compile time. An example of schema definition is presented in Source Code 12, where the schema for a set of poems from the italian poet and journalist Eugenio Montale is shown. Each textual object declaration is composed by a name, a list of components and their types, a list of annotations and their type, the name of its plural associated type, and list of annotations for the plural type and their types. A type can inherit all the characteristics of another type through the keyword *inherits*. Declaration blocks are usually given only in programs that parse an input text to instance a textual repository. When the instantiation is done, the information about types are also stored in the repository. In this way, when a program connects to the repository to analyze or annotate the text, all type

information are retrieved without the need of further declarations[1].

Source Code 12 Example of schema definition for Eugenio Montale's poems; type names are in italian.

```
declare schema montale
        type PAROLA = textualobjecttype
                attributes stem : Fun():String is stem_of(self.underlying_text)
        plural PAROLE,
        plural attributes comment : String
        end

        type VERSO = textualobjecttype
                components parole : PAROLE
                attributes metrica : String
                plural VERSI
                plural attributes comment : String
        end

        type FRASE = textualobjecttype
                component parole : PAROLE
                plural FRASE
        end

        type TITOLO = textualobjecttype
                inherits FRASE
                plural TITOLI
        end

        type STROFA = textualobjecttype
                components versi : VERSI, frasi : FRASI
                plural STROFE
        end

        type POESIA = textualobjecttype
                components strofe : STROFE, titolo : TITOLO
                attributes dedica : String
                plural POESIE
        end

        type LIBRO = textualobjecttype
                components poesie : POESIE, titolo : TITOLO
                attributes autore : {nome : String, cognome : String}
                plural POESIE
        end

        type COLLECTION = textualobjecttype
                components libri:LIBRI
                attributes titolo : String
        end
end
```

For what concerns the declaration of identifiers, instead, the Manuzio language uses the notion of environment, a map from identifiers to definitions of types or values. Environments can be extended with new pairs at runtime through declarations. A declaration is an expression that returns a *nop* and, as a side effect, adds to the environments a new bind between an identifier and a type (in the case of a

[1]See Section 5.5.20 for further information about connecting to a repository.

type declaration) or a value (in the case of a constant declaration). An example of declaration's usage can be found in Source Code 13.

Note that textual object types declarations are available only in the schema declaration block of the language, and are otherwise considered invalid expressions. This restriction is useful to define two distinct contexts, or program classes: one where the textual schema is defined and instanced, and one where the connection with a textual repository is performed to query and annotate the contained textual data.

Source Code 13 Example of declarations in Manuzio.

```
1 type N = Int;
2 let n:N = 0;
3 let another_n:Int = n;
```

4.3.4 Numbers

In Manuzio numbers are represented by integers and reals. Integers are equipped with the classical binary arithmetic operators $(+, *, -, /)$ and with the unary minus to for negative values. Also the classical relational operators are present. Since to deal with numbers is not a central topic in Manuzio only these basic integer operations has been included so far. If the need of a more complete set should arise it will be easy to add the required operations. A division by zero raise a runtime error and cause the stop of the current computation. The same set of operators is valid for real numbers. Manuzio does not, at the current time, support implicit conversions between integers and reals, so a cast operator (discussed later) must be explicitly called. A cast performs a round of the real number, but a truncate operator is also present. Integer constants are denoted by the equivalent integer number, while real constants are denoted by a their integer part followed by a dot and by their decimal part. In Source Code 14 an example of usage of some basic operators is given, while in Table 4.1 the complete list of operators on those types is presented.

Source Code 14 Integers and reals usage.

```
1 let n : Int = 1;
2 n + 4;
3         #=> 5 : Int
4 let r : Real = 2.5;
5 let s : Real = 3.0;
6 2.5 / 3.0
7         #=> 0.833333333333333 : Real
```

	Integer Operators	Real Operators
Arithmetic	$+, -, *, /$	$+, -, *, /$
Relational	$>, \geq, <, \leq, =$	$>, \geq, <, \leq, =$
Others		*trunc*, *round*

Table 4.1: Numerical operators in Manuzio.

4.3.5 Booleans

Manuzio has a specific type to represent boolean values. The boolean type is a set of two values that are denoted, in our language, with the literals *true* or *false*. As with numbers only a limited, basic set of operators on boolean values is present, with the same motivations. Booleans can be tested for equality, negated, or combined with an *AND* or *OR* logic. Boolean values can be declared directly but are more often obtained as the result of a relational operator. Examples of booleans usage is given in Source Code 15, while in Table 4.2 a complete list of booleans operators is shown.

Source Code 15 Booleans usage.

```
1 true;
2         #=> true : Bool
3 NOT true;
4         #=> false
5 r > s
6         #=> true : Bool
7 let b : Bool = false;
8 b AND (r>s)
9         #=> false : Bool
```

Boolean Operators
$AND, OR, NOT, =$

Table 4.2: Boolean operators in Manuzio.

4.3.6 Strings

The string type is a type of a certain importance in Manuzio since it is the data type normally used to contain texts. In Manuzio strings are a primitive type with their own set of operators. String literals are denoted by enclosing an arbitrary long list of characters in double quotes and can contain escape characters to control formatting.

Strings can be compared with relational operators to obtain their lexicographic order, concatenated, or repeated n times. Strings in Manuzio can also be matched

with a regular expression with the *match* operator. A regular expression literal is prepended by the *regexp* keyword and delimited by forward slashes. Regular expressions are used to describe a textual pattern and follow closely the syntax of Ruby ones. For brevity regular expressions syntax rules are not reported here, but an introduction can be found in [Matsumoto and Ishituka, 2002], while a book-length coverage can be found in [Friedl, 2006].

Examples of strings usage is given in Source Code 16, while in Table 4.3 a complete list of strings operators is shown.

Source Code 16 Strings usage in Manuzio.

```
1  "manuzio" + " is slow " + "but good"
2          #=> "manuzio is slow but good" : String
3
4  "manuzio " * 3
5          #=> "manuzio manuzio manuzio " : String
6
7  "manuzio" slice (1..3)
8          #=> "man" : String
9
10 let r = regexp /[mM]anu.*/
11         #=> regexp /[mM]anu.*/ : Regexp
12
13 "manuzio" ~ r
14         #=> true : Bool
15
16 "Manubrio" ~ r
17         #=>true : Bool
18
19 "MANUZIO" ~ r
20         #=> false : Bool
```

	String Operators
Comparison	$>, \geq, <, \leq, =$
Manipulation	$+, *$
Matching	$\sim$

Table 4.3: String operators in Manuzio.

4.3.7 Variables

Even tough Manuzio is in principle a functional language one of its bundles include the non-purely functional notion of memory and memory locations as variables. A variable is a location in memory that can be updated by an assignment operation. In Manuzio the type of an updateable value of type T is denoted by *vartype* T, while a variable value is denoted by the value itself prepended by the keyword *var*. The special operators relative to variables are the *assign* operator, denoted by a

colon followed by an equal sign, that is used to store a variable value in a location of memory, and the *at* operator, denoted by an exclamative mark, that is used to retrieve a value from a memory location.

Manuzio has also the concept of unknown value. The *NULL* type is a subtype of any other type, so that its singleton value *nil* can be stored into memory locations of any type to denote an unknown or invalid value that can be later updated with a valid one. When an unknown value must be evaluated the evaluator returns a runtime error.

4.3.8 Functional Abstractions

In Manuzio functions are first-order values, so that a function can be the result of an expressions, can be passed as parameter, and so on. Function values are denoted by the *fun* keyword, while function application is, differently from other languages, denoted by the @ symbol followed by the name of the function. The value of a function is a closure. Manuzio employ the static binding technique, so that a function body is evaluated in an environment formed by the values of the actual parameters and the values of other constants at the time of the function definition. For instance, in Source Code 17 the value of the function application does not change when the value of *a* changes.

Source Code 17 Functions usage.

```
1 let a = 0;
2 let f = fun(x:Int):Int is x+a;
3 @f(0)
4         #=> 0 : Int
5 let a = 1;
6 @f(0)
7         #=> 0 : Int
```

Recursive functions need a different syntax to be declared. A recursive function is denoted by the *recfun* keyword followed by the name of an identifier that will be used for self-reference in the body of the function. In Source Code 18, for instance, a recursive function to compute the factorial of an integer number is declared as *fact*[2], with the self-identifier *me*.

Source Code 18 Recursive functions usage.

```
1 let fact = recfun me(n:Int):Int is if n=1 then 1 else n*@me(n-1) end
2 @fact(4)
3         #=> 24 : Int
4 @fact(5)
5         #=> 120 : Int
```

[2]For simplicity, we assume the parameter to be greater then zero in this context.

Manuzio supports the declaration of polymorphic functions. A polymorphic function is a function that takes types as parameters and returns a function as result. Such functions are declared with the *polyfun* keyword followed by the parametric types identifiers enclosed in square brackets. A polymorphic function can later be instantiated by using the application operator @ followed by the the actual types. The returned value is a function on those types that can be later applied to values. For instance, in Source Code 19, the identity function is declared and then applied to integer and boolean values. A polymorphic function with two type parameters is also declared. Since parametric types consistency is also enforced by the type checker, the last line of the code yields to a type error, since the return type and the function body type are not compatible after the polymorphic function has been instanced.

Source Code 19 Polymorphic functions usage.

```
1  let id = polyfun [X] is fun(x:X):X is x
2  let idInt = @id[Int]
3  @idInt(0)
4          #=> 0 : Int
5
6  let idBool = @id[Bool]
7  @idBool(false)
8          #=> false : Bool
9
10 let f = polyfun [X,Y] is fun(x:X, y:Y):X is y
11 let g = @f[Int, Int]
12 @g(0,1)
13         #=> 1 : Int
14
15 let u = @f[Bool, Bool]
16 @u(true,false)
17         #=> false : Bool
18
19 let u = @f[Int, Bool]
20         #=> _TypeClashError_
```

4.3.9 Records

The *record* data structure consists in a set of pairs $(label, value)$, where the order is not important. Records are equipped with the *dot* operator to access their fields through label names, with the *extend* operator to create a new record with additional fields, and with the *project* operator to create a new tuple with less fields than the original one.

The program in Source Code 20 shows an example of record usage in Manuzio. A record type `POINT` is defined along with an object of that type named p. Then another type, `COLORPOINT` is defined and an instance of it is declared as an extension of p. The *shift* function takes a record as parameter and returns a new record built by shifting the parameter coordinates. The project operator is then used to return another new, narrowed, record.

Source Code 20 Records usage.

```
type POINT = {x:Int, y:Int};
        #=> nop : Command
let p:POINT = {x=0, y=0};
        #=> nop : Command

type COLORPOINT = {x:Int, y:Int, color:String};
        #=> nop : Command
let pc:COLORPOINT = p extend {color="Black"};
        #=> nop : Command

let shift = fun(p:POINT, dx:Int, dy:Int):POINT is {x=(p.x + dx), y = (p.y +dy)};
        #=> nop : Command

@shift(p, 1, 1);
        #=> {x=1, y=1} : POINT

let shifted_p = @shift(p, 1, 1);
        #=> nop: Command

shifted_p project {x};
        #=>{x=1} : {x:Int}
```

4.3.10 Sequences

Sequences are important values in functional languages since they cope well with recursive algorithms. Manuzio features a sequence type constructor equipped with a rich set of operators. A sequence is denoted by a list of values, separated by commas, all enclosed in square brackets. A sequence's type is the most general type among its element's types. This means that the elements of a sequence do not need to be of the same type, but their types must be at least compatible with each other. Empty sequences exists in Manuzio, but since the type checker needs to know the type of such sequences too it is necessary to declare them with the *emptyseqof* keyword followed by a type name, rather then with the simpler *[]*. Sequences are equipped with classical operators to extract the *head* of the sequence and to return its *tail*. The first operator returns an element of the same type of the sequence's elements type, while the second returns a new sequence of the same type. The *cons* operator return a sequence with a new value inserted in the head position, while the *append* operator is used to concatenates two different sequences. Sequences can also be treated as multisets by the use of the *intersect*, *union*, and *difference* operators. Examples of such operators usage can be found in Source Code 21.

A number of other, more specific, sequence operators exists in Manuzio. The *flatten* operator performs a flattening of sequence of sequences values, so that the returned value is a one-level sequence of values. The *isin* operator, instead, returns a boolean value dependent on the presence of a value as at least one of the list elements. In the same way the *isempty* operator returns true if an empty sequence is passed. Examples of such operators usage can be found in Source Code 22.

Finally, the *in* operator constructs a sequence of records from a sequence of values. This operator usage will be discussed later when it will be used in conjunction

Source Code 21 Sequence basic operators usage.

```
let s =[0,1,2,3]
head s
        #=> 0 : Int
tail s
        #=> [1,2,3] : [Int]
-1 cons s
        #=> [-1,0,1,2,3] : [Int]

emptyseq of Int
        #=> [] : Int
0 cons emptyseq of Int
        #=> [0] : [Int]

[{x=0,y=0}, {x=1,y=1}, {x=0, y=0, z=0}]
        #=> [{x=0,y=0}, {x=1,y=1}, {x=0, y=0, z=0}] : [{x:Int, y:Int}]

[1,1,2] union [1,2,2,3]
        #=> [1,1,1,2,2,2,3] : [Int]

[1,1,2] intersect [1,2,2,3]
        #=> [1,2] : [Int]

[1,1,2] difference [1,2,2,3]
        #=> [1] : [Int]
```

Source Code 22 Sequence basic operators usage.

```
flatten [[0,1],[1,2],[3,4]]
        #=> [0,1,1,2,3,4] : [Int]

0 isin [0,1,2,3]
        #=> true : Bool
```

with query-like operators and discussed later.

4.3.11 Objects

Manuzio have a simple object model that takes inspiration from the work presented in [Albano and Procopio, 1998]. An object is an entity with a state, an identity and a behavior. The *state* is a set of values called *properties*, the *behavior* is a set of local procedures, called *methods*. The *identity* of an object is a time-invariant property which function is to make every object different from each other. Two objects with the same properties values but with different identity are thus different. An object can receive messages to which it responds by returning one of its properties values or the return value of a method. Objects have types, and the type of an object is also called its *interface* because it lists the messages that the object can accept. Manuzio objects are voluntarily simple because their goal is to be used as the foundation in the design of textual objects. They lack some advanced features present in other, classical, object-oriented languages that we felt were not useful or interesting when applied to a textual analysis context.

Manuzio is an *object-based*, or prototype based, language. In object-based programming classes are not present and behavior reuse is performed via a process of cloning existing objects that serves as prototype, in a way similar to the class-based languages inheritance one. An object can be *extended* with new properties to create a new object. The choice of an object-based language has been made mainly to achieve simplicity in implementation and for the freedom of not being constrained by a class structure. Future implementations could, if needed, introduce other concepts like class and encapsulation without breaking the textual model.

Source Code 23 Example of objects usage in Manuzio.

```
type CELL = OBJECT(
   {contents:vartype Int,
        getContents:methodtype():Int,
        setContents:methodtype(Int):Command
   }
);

let cell:CELL = object(
   {contents=var 0,
        getContents = method():Int is !self.contents,
        setContents = method(n:Int):Command is self.contents := n
   }
);

type RECELL = OBJECT(
        {contents:vartype Int,
         backup:vartype Int,
         getContents:methodtype():Int,
         getContents:methodtype(Int):Command,
         restore:methodtype():Command
        }
);

let reCell:RECELL = cell objectextend {
        backup = var !self.contents,
         getContents = method(n:Int):Command is begin
                self.backup := !self.contents;
                self.contents := n;
         end,
         restore = method():Command is self.contents := !self.backup
};

reCell.@set(5);
let a = reCell.@getContents();
reCell.@restore();
let b = reCell.@getContents();
```

In Manuzio, object types are denoted by the *OBJECT* keyword, followed by a record of instance variables and methods enclosed in parenthesis. Object instances, instead, are denoted by the keyword *object*, followed by a record value enclosed in parenthesis. In the object definition the special identifier *self* is used to refer to the object itself. Inside an object type the keyword *methodtype* is used to denote methods signatures, while the keyword *method* is used to denote a method definition inside an object value. The program in Source Code 23 shows an example of objects

usage.

In object types only method signatures are specified, while method bodies are contained in the single instances definitions. For this reason, in Manuzio, two objects of the same type can implement a method in two different ways, as long as the implementation is consistent with the specified signature. This behavior is called *multiple implementations* of objects. To create objects with the same method implementation it is possible to define a function that works as a *constructor*. A constructor takes in input the values required to set an object instance variables with and returns an object. For instance, a constructor for the `CELL` type of Source Code 23 can be defined as shown in Source Code 24, where a cell constructor is used to instantiate a cell constant containing the integer value of zero.

Source Code 24 Example of an object constructor.

```
let createCell = fun(n:Int):CELL is
        object(
           {contents=var n,
                getContents = method():Int is !self.contents,
                setContents = method(n:Int):Command is self.contents := n
           }
        )
let cell : CELL = createCell(0);
```

4.3.12 Textual Objects

The Manuzio language features a peculiar data type called *textual objects*. With "textual object" we refer to both single and repeated textual objects. The precise notion of textual object has been defined in the Manuzio model, presented in Section 3.

A single textual object is a special kind of object with some unique characteristics. Textual objects are used to represent a portion of a text with a logical meaning. Textual objects are composed by a state, a behavior, and a set of relationships with other textual objects. The state is represented by a collection of *instance variables*, values of any type that, when applied to textual objects, are also called *attributes*. The behavior is represented by a set of local procedures called *methods*. Finally, the relationships with other textual objects are described by a set of values called *components*. Each of these values can be generically referred to as a *field* of its textual object.

Repeated textual objects, instead, are an homogeneous repetition of textual objects. Repeated textual objects, also called repetitions for brevity, can be seen as a sequence of textual objects of a certain type with the addition of a set of associated values as *attributes* and a set of methods. The textual objects that constitute a repetition are called its *elements*.

When a *message* is sent to a textual object, the corresponding method is executed. Since encapsulation is not required nor defined in the model, asking directly

for the value of an attribute or component is also considered a valid message. Textual objects are equipped with a set of predefined operators, the most important is the *dot* operator used to access its fields. Another family of operators, called the *component access operators*, are used to access components. Such operators are used to traverse the textual hierarchy by exploiting the component relation among textual types defined in the model. Such operators are the *get..of* and the *getall..of* operators.

The get operator behave just like direct component access. By specifying the name of a component the textual object (or repeated textual object) associated to that component is returned. Textual objects operators works in a seamless way on both single and repeated textual objects. When applied to repeated textual objects the get operator flatten its results to a single repeated textual object which elements are a union of the corresponding mapping elements. This behavior has been chosen because it represents the most common way of accessing data in a literary text context. A mapping can instead be obtained by applying the *collect* operator of the language[3]. The get operator can be used in the same way to access attributes. In this case the semantics, when applied to repetitions, is exactly the one of a mapping.

Source Code 25 Textual objects usage in an italian songs textual database.

```
1  get title of collection
2          #=> "Collection of Songs from F.Guccini" : String
3
4  let p1 = head get poems of collection
5  let p2 = head tail get poems of collection
6
7  text of get title of p1
8          #=> "Il Vecchio e il Bambino" : String
9
10 text of get title of p2
11         #=> "Canzone delle Domande Consuete" : String
12
13 let w = getall WORD of p2
14 text of w
15         #=> ["Ancora", "qui", "a", "domandarsi", ... "te"] : [String]
```

The getall operator is used to access both direct and indirect components of a textual object. By specifying a textual type T the getall operator traverse the hierarchy of types starting from the caller object t and fetch all the textual objects of type T that are direct or indirect components of t. The resulting value is a repeated textual object which elements are of type T that contains the results. Note that, since there can be multiple paths connecting the type T with another type S, one component can be found multiple times. This is not an issue because as stated in definition 5 in Section 3, all the elements of a repetitions are unique, so a set union of the results is performed.

[3]It is important to note that the collect operator is not a textual objects specific operator, but can be applied to textual objects without issues because textual objects are a native type of the language.

Another important operator on textual objects is the *text of* operator, that performs the mapping between textual objects and their underlying text. Such operator returns a string, and can be applied both to single textual objects and to repetitions. When applied to repetitions the operator returns a sequence of strings, each containing the text of one of the repetition's elements. An example of component access and text operators usage can be found in Source Code 25.

Starting from a textual object t the operator *parent of* can be invoked with a textual type T parameter to retrieve the object of type T that is, directly or indirectly, a parent of t. Note that the component relation definition implies that this object is always unique. Textual objects are ordered following the natural ordering of text. For this reason, given two textual objects of the same type, we can compute their distance as the number of textual objects of the same type that separate them, plus one. Two object whose distance is 1 are said to be contiguous. The distance is computed by the *distance from* operator; its results is a positive integer if the first argument occur after the second, negative vice-versa. An object have distance 0 from itself. An useful operator, derived from the distance operator, is the *surround* operator, that, given a textual object t and a positive integer number n, returns the repeated textual object composed by t and all the objects at distance n or less from it.

Textual objects are also equipped with a set of operators for *textual comparison*. Such operators can be used to test for various properties of a textual object underlying text. The semantics of these operators is based on the definition 10 presented in Section 3. Such operators take into account the natural linear order of their underlying text. The language has operators to test the relations between object's positions, known as the Allen's relations[Allen, 1991]. They allow, for instance, to know if an object is fully contained in another one, if one partially precedes another, and so on[4].

4.3.13 Persistence and Query Operators

A persistent programming language seamlessly allows values to continue existing after the program has been terminated. In such languages the persistence is transparent, differently from other techniques, like `SQL` embedding, where the programmer must make two different paradigms coexist.

One of the oldest programming language with persistent capabilities has been `MUMPS` [Wasserman and Sherertz, 1976], a language created in the 1960s and still in use in the field of medical records keeping, and also widely used in banking. In `MUMPS`, all global lists are automatically persisted. Other notable examples of such languages include Napier88 [Morrison et al., 1999] and Persistent Modula-3 [Hosking and Chen, 1999]. A large number of external libraries to perform some

[4]Since the underlying text of objects may be non-contiguous, complex situations can arise. It is sufficient to say here that the comparison operators are always well defined for every possible pair of objects.

sort of persistence (most notably object-relational mappings) on other languages also exists, like Ruby's ActiveRecord or Java Hibernate.

While Manuzio is not a persistent programming language, it shares some characteristics of such languages when handling textual databases and textual objects. When a textual database that follows the Manuzio model is available it can be accessed by issuing the *use database* command to the interpreter. Such command checks the effective availability of the database and its conformance with the Manuzio model, and then, if everything is fine, creates a link between the database and the language. When such link is created the Manuzio language can handle textual objects as native data types. The textual types contained in the database are added to the environment and a special value, called *collection* is created to refer to the persistent collection root. Textual objects are read from the textual database without any need from the coder to write in a language different from the Manuzio language or to deal with data type mismatching like it would have been necessary with `SQL` embedding. Everything is done in an high-level way, so that the language user does not need to know about the persistent layer implementation to use it.

The language has specific operators to query the textual database, called *query-like operators*. Some of these operators are inspired by the ones present in the Galileo language[Albano et al., 1985]. The syntax is similar to the one typical of relational databases query languages. Since textual objects are values of the language, query-like operators in Manuzio can be applied to both sequences and repeated textual objects with exactly the same logic.

The key operator when discussing query-like operators is the *in* operator. It takes two arguments, an identifier *id* and a sequence or a repeated textual object, and returns a sequence of records in the form $[\{id = e_1\}, ...\{id = e_n\}]$, where e_i is the i-nth element of the input sequence or repetition. The *in* operator is overloaded to work on both sequences and repeated textual objects alike.

The following constructs are available on sequences of records:

- (*s* **where** *b*): returns the sequence of the values that satisfy the boolean expression *b*.
- (**some** *s* **with** *b*): tests whether at least one element in *s* satisfies *b*.
- (**each** *s* **with** *b*): tests if all elements of *s* satisfies *b*.
- (**select** *e* **from** *s*): returns the sequence of the values of the expression *e* evaluated for each element in a sequence *s*.
- (*s* **extend*** $\{id_1 : T_1 = v_1, \ldots, id_n : T_n = v_n\}$): returns the sequence of values obtained by applying the operator extend to each element in *s*.
- (*s* **project*** $\{id_1 : T_1, \ldots, id_n : T_n\}$): returns the sequence of values obtained by applying the operator project to each element in *s*.

- (s **groupby** $\{id_1 = v_1, \ldots id_n = T_n\}$): returns a sequence with type $[\{id_1 : T_1, \ldots, id_n : T_n, partition : [T_s]\}]$. The elements of partition are those of s with the same value of $\{id_1 = v_1, \ldots id_n = T_n\}$, which is evaluated for each element e of e in the current environment extended with the fields of e. The results are evaluated as follows:

  1. For each element e of s the record $\{id_1 = v_1, \ldots, id_n = v_n\}$ is evaluated in the current environment, extended with the fields of e, to produce a sequence w of pairs $[e, \{id_1 = v_1, \ldots, id_n = v_n\}]$.
  2. The elements of w are partitioned in subsequences that shares the same value of $\{id_1 = v_1, \ldots, id_n = v_n\}$. We call $[e_i]$ the sequence of the fields e of the elements of a partition.
  3. A sequence of one element for each partition is produced. Such elements are in the form $\{id_1 = v_1, \ldots, id_n = v_n, partition = [e_i]\}$.

where s is a sequence of records, b is a boolean expression, id_i is an identifier, and e is an expression. Some simple examples of the use of these operators can be found in Source Code 26, where a program to compute and show all the words with a specific suffix in different songs is given. Other query examples are shown ad explained in more detail in Section 6.5.

Source Code 26 Example of a program to retrieve a set of similar words from two different songs.

```
#connecto to the guccini database
usedb "guccini";

let collectionTitle:SENTENCE =  get title of collection;
output "Analyzing collection: " + (text of collectionTitle);

let song1:SONG = head (get songs of collection);
let song2:SONG = head tail get songs of collection;

output "Analyzing song: " + (get title of song1);
let words = getall WORD of song1;

let areWords = select word
                           from (word in words)
                           where (text of word) ~ regexp ".*are$";
output "In " + (get title of song1) +
          " the words that finish in -are are " + (size of areWords)%String +
          ": " + (text of areWords);

output "Analyzing song: " + (get title of song2);
let words = getall WORD of song2;

let areWords = select word
                           from (word in words)
                           where (text of word) ~ regexp ".*are$";
output "In " + (get title of song2) +
          " the words that finish in -are are " + (size of areWords)%String +
          ": " + (text of areWords);
```

Lazy Evaluation and Query Composition

One of the drawbacks of persistent languages is that it is hard to do automatic optimizations. Accessing persistently stored data is a costly operation and the access overhead could hinder the performances of such approach. This problem very important in Manuzio because even a medium sized text corpora, analyzed at word-level, is composed by a large number of textual objects[5].

We discussed lazy evaluation in the context of functional programming languages earlier in Section 4.2. A modified version of lazy evaluation can help performances by delaying the execution of queries to the last possible moment [Zi et al., 1992]. We plan to apply lazy evaluation only to expressions involving textual objects, that is, expressions with the need to access the persistent textual database. The result of such operators is a special kind of function called *thunk*. A thunk is a non-evaluated value that knows how to retrieve the needed data from the textual database, for instance by issuing an `SQL` query to a relational database, assuming our system uses this technology for storage.

If a thunk result is needed, for instance to display the results, the language evaluate it and materialize its actual value. Instead, if such results must be further manipulated, lazy evaluation can be exploited to perform *query composition*. With query composition the programmer can avoid to write complex commands all together and, instead, write a chain of simpler language statements that yield the same result in roughly the same time. Moreover, by using memoization techniques discussed earlier, the persistence layer can store the results of queries for future reuse.

The query composition implementation is heavily dependent on the solution adopted for storage. In our prototype the use of a relational database made possible to express some operator chaining in `SQL`, but a more complete and useful solution would be achieved by the use of an ad-hoc storage system.

Other languages adopt a similar behavior. Microsoft LINQ [Meijer et al., 2006], for instance, allows the programmer to express `SQL` queries directly from the language as Manuzio do with textual objects. Their extensive work on adapters made possible, however, to apply this idea to a wider range of data types and operators. While not needed in the toy examples presented in this thesis, query composition will be a central topic when developing real-world applications, where not only the textual objects cardinality will be higher but performances will be a more concrete problem. While open problems exists for the implementation of this solution, some possible implementation principles are discussed in Section 6.6.

[5]The 43 Shakespeare's works, for instance, contains 884,429 words in 34,896 lines.

5

The Manuzio Language Semantics

"*Good code is its own best documentation.*" – Steve McConnell

5.1 An Introduction to Operational Semantics

The Semantics of a language defines the meaning of programs written in that language, how they behave when they are executed. Often different languages use different syntax and different semantics to express similar constructs. While variations in syntax are in general superficial, semantic differences express a difference in the meaning of the program. As a consequence, similar constructs with different semantics may produce very different results.

The semantics of programming languages can be defined by two different steps: *static semantics*, also called *typing*, and *dynamic semantics*, also called just semantics.

Static Semantics

Types are part of the semantics of the language, called its static semantics. The goal of static semantics is to detect programs that will give errors during execution before the actual execution of such programs. Note that, since static semantics checks take place after the syntax analyzer, we assume that programs are, at this stage, syntactically correct. When an expression of the language has a defined type it is said that it is *typeable*. The process of checking for consistency of static semantics rules is called *typechecking*. The set of rules that compose the static semantics of a language are called its *type system*. A type checking rule can be in one of two forms. A rule in the form:

$$\varepsilon \vdash e : T \tag{5.1}$$

indicates that with the types specified in the type environment ε, the expression e has type T. A rule in the form:

$$\frac{\varepsilon_1 \vdash e_1 : T_1, \ldots, \varepsilon_n \vdash e_n : T_n}{\varepsilon \vdash e : T} \tag{5.2}$$

indicates that with the typing in ε, expression e has type T as long as the assertions above the horizontal line all hold. The assertions over the line are called *hypothesis* while the assertion below the line is called the *conclusion*. In the rest of the thesis the following abbreviation will be used, if two hypothesis shares the same type environment, the repetition of the type environment can be omitted and the associated assertions are separated by commas, as in:

$$\frac{\varepsilon_1 \vdash e_1 : T_1, \ldots e_n : T_n \ldots, \varepsilon_{n+1} \vdash e_{n+1} : T_{n+1}}{\varepsilon \vdash e : T} \tag{5.3}$$

where the first n hypothesis are valid in the ε_1 environment, while the $n+1$th one is valid in the ε_{n+1} one.

We will use such formalism to express type-checking rules and define the set of legal expressions of our language.

Dynamic Semantics

The dynamic semantics of a language describes the behavior of programs written in that language at execution time. Often it is given by informal definitions and examples, but such way of defining dynamic semantics can be incomplete and ambiguous. The right way of defining a language semantics is through formal specification, for several reasons:

- Language Implementation: a formal semantics facilitates the implementation of compilers and interpreters, due to a clear and machine independent specification of each language construct.
- Programming: a clear description of the language constructs meaning can, along with examples, ease the learning curve of a new language to programmers.
- Language Design: in the process of language design a formal semantics specification can aid to avoid inconsistencies and suggest improvements.

There are several approaches to specify formal semantics of programming languages, the main ones are:

- Denotational: in denotational semantics mathematical objects, called denotations, are used to describe the meaning of expressions from the language in an abstract way.
- Axiomatic: in axiomatic semantics the meaning of programs is given by describing the properties that hold before and after the execution of the program, using axioms and deduction rules in a specific logic. Each construct of the language is constrained by some preconditions and postconditions.

- Operational: operational semantics describes how a valid program is interpreted as a sequence of computational steps. Transition systems are used as a tool to give operational specification: the execution of the program is described as a sequence of transitions in an abstract machine, often a state machine. Axioms and rules are used to define the possible transition relations of such machine.

Each formal definition of semantics has its advantages. Operational semantics, however, has been proven to be of particular practical use during the implementation of a language compiler or interpreter, since it describes the computational steps required to execute each expression. For this reason the Manuzio language presented in this thesis will be given a formal specification with operational semantics.

Mathematical Foundations of Operational Semantics

Operational semantics is based on the concept of *transition system*. A transition system is a mathematical entity that can be used to model computation.

Definition 13 ***(Transition System)*** *A transition system is specified by:*

- *a set* S *of states;*
- *a binary relation* $\rightarrow \subseteq S \times S$ *called the transition relation.*

We use the notation $s \rightarrow s'$ to indicate that s and s' are in such relation. A transition relation can be interpreted as a change of state from s to s'. We denote by $\rightarrow *$ the reflexive closure of the transition relation, so that $s \rightarrow *s'$ hold if and only if there is a sequence of transitions:

$$c \rightarrow c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_n \tag{5.4}$$

where $n \geq 0$ and $c_n = c'$.

We distinguish two subsets of S, I and T, containing initial and final configurations. A final configuration is a configuration s with no possible transitions out, so that there is no s' for which $s \rightarrow s'$ exists. The intuitive idea is that a sequence of transitions from an initial state $s \in I$ to a final state $s' \in T$ represents the run of a program.

Definition 14 ***(Deterministic Transition System)*** *A transition system is deterministic when for every state s, if $s \rightarrow s'$ and $s \rightarrow s''$ then $s' = s''$.*

In a deterministic transition system at each point of computation there is only one possible transition. By specifying a transition system it is possible to give a formal semantics to a programming language. The system is seen as an abstract machine

that simulate the execution of a program that can be used as a model to implement an interpreter for the programming language.

While an abstract machine is useful to implement interpreters since it describe the execution step by step, it include usually too much details to be easily understood by users. Another way to give a language an operational semantics is through *structural operational semantics*, which is based on transition systems and gives an *inductive* definition of the execution of programs. The transition relation is defined by induction and each command of the language is described in terms of its components.

By using the mathematical notion of induction applied to trees[Fernandez, 2004] we can define subsets of a set T. We write inductive definitions as *axioms*, that represent the base of induction, and *rules*, that represent inductive steps.

Definition 15 ***(Axioms and rules)*** *An axiom is an element of the final states set T. A rule is a pair (H, s) where:*

- *H is a non-empty subset of T, called the* hypothesis *of the rule;*
- *s is an element of T, called the* conclusion *of the rule.*

Definition 16 ***(Inductive Set)*** *The subset I of T inductively defined by a collection A of axioms and a set R of rules consists of those $t \in T$ such that:*

- *t is an axiom, or*
- *there are $t_1, t_2, \ldots, t_n \in I$ and a rule $(H, s) \in R$ such that $H = \{t_1, \ldots, t_n\}$ and $t = s$.*

To show that an element t is in the set I it is sufficient to show that t is an axiom or that exists a proof tree with root in t. A proof tree is a tree of rules where the root is the conclusion and the leaves are axioms. Such trees are written in the following way:

$$\frac{\dfrac{\dfrac{\vdots}{t_{11}}\cdots\dfrac{\vdots}{t_{1m_1}}}{t_1}\cdots\dfrac{\dfrac{\vdots}{t_{n1}}\cdots\dfrac{\vdots}{t_{nm_n}}}{t_n}}{t} \tag{5.5}$$

where for each non-leaf node t_i there is a rule $(\{t_{i_1}, \ldots, t_{i_{m_i}}\}, t_i)$.

We can define an evaluation relation for a programming language using axioms and rules applied to an abstract syntax tree of expressions. We will write $e \Rightarrow v$ to indicate that the expression e evaluate in the value v. An axiom is denoted by the following scheme:

$$e \Rightarrow v \tag{5.6}$$

and represent an unconditioned evaluation. A rule is represented as:

$$\frac{e_1 \Rightarrow v_1, e_2 \Rightarrow v_2}{+(e_1, e_2) \Rightarrow v_1 + v_2} \tag{5.7}$$

meaning that if the expression e_1 evaluates in the value v_1 and the expression e_2 in the value v_2 then the expressions $+(e_1, e_2)$ evaluates in the sum of v_1 and v_2.

Usually evaluation takes place in a certain environment Γ, a set of pairs that bind an identifier to a value. In this case axiom and rule schemes change to reflect the presence of this structure.

$$\Gamma \vdash e \Rightarrow v \tag{5.8}$$

meaning that in the environment Γ the expression e evaluates in the value v without conditions, while:

$$\frac{\Gamma_1 \vdash e_1 \Rightarrow v_1, \Gamma_2 \vdash e_2 \Rightarrow v_2}{\Gamma \vdash +(e_1, e_2) \Rightarrow v_1 + v_2} \tag{5.9}$$

means that if in the environment Γ_1 the expression e_1 evaluates in the value v_1 and in the environment Γ_2 the expression e_2 in the value v_2 then in the environment Γ the expressions $+(e_1, e_2)$ evaluates in the sum of v_1 and v_2. If two or more hypothesis shares the same environment then repeated environment declarations will be omitted in the rest of the thesis, like in:

$$\frac{\Gamma_1 \vdash e_1 \Rightarrow v_1, e_2 \Rightarrow v_2}{\Gamma \vdash +(e_1, e_2) \Rightarrow v_1 + v_2} \tag{5.10}$$

meaning that both e_1 and e_2 have the indicated value when evaluated in Γ_1.

5.2 Formal Language Specification

The Manuzio programming language is a prototype language enriched with special constructs to assist the writing of programs that needs to analyze, annotate and store persistently textual data, with a special focus on literary texts. The main features of Manuzio are:

- Manuzio is a strongly statically typed language: each expression of the language has a type and each sentence is type checked before being executed.
- Manuzio is an interactive language: each sentence of the language is an expression and each expression has a type and produces a value.

- Manuzio is a functional language: functions are first order objects, that can be passed as parameters, returned by other functions or be used in data structures.

Analyzing this language will be the main focus of this section. We will first carefully give formal specifications of the syntax, type-checking rules, and semantics of Manuzio. With that specifications we will be able to introduce its innovative features in a painless, sound way. In order to make simple to write down semantics and type-checking rules for our language the syntax will be somewhat more rigid and less elegant than other, more advanced, object-oriented languages. Abbreviations and syntactic sugar can be easily applied in subsequent time.

Manuzio has been developed in an modular way. We will start by describing only the general rules of the language, without referring to specific data types or values like integers, functions, strings and so on. We will call this empty language μManuzio, and while it will have no expressive power (you can't write anything more then the empty program with it) it will be the platform over which the extensions, called *bundles* will be specified and developed. Each bundle of the language can enrich the language with its own values, types, type-checking rules, and expressions. Each of these aspects will be defined in each of the bundles subsection.

5.2.1 Bundle Dependency Graph

In Manuzio bundles are not independent. For instance when we defined the `Integers` bundle we made use of the *Bool* type to type check the relational operators between integers, a declaration needs an identifier to be written, and so on. To make things clear we draw a graph of the dependencies between the proposed bundles. An arrow from a bundle b to a bundle b' means that b' needs b to work.

In some case even if there is not a dependency between two bundles, they can be strongly tied. For instance, it is possible to write a correct program that makes use of iteration without variables, or to use integer numbers without anything else than booleans, but the resulting programs would be trivial at least.

5.3 Types

Manuzio is a strongly, statically typed language. Types provide an implicit context for many operations, and limit the set of operations that may be performed in a semantically valid program. Informally, a *type system* consists of a mechanism for defining types and associating them with certain language constructs and of a set of rules for *type equivalence*, *type compatibility*, and *type inference*. In Manuzio each construct of the language has a value, and each value has a specific type. Type equivalence rules define when two types can be considered the same type. Type

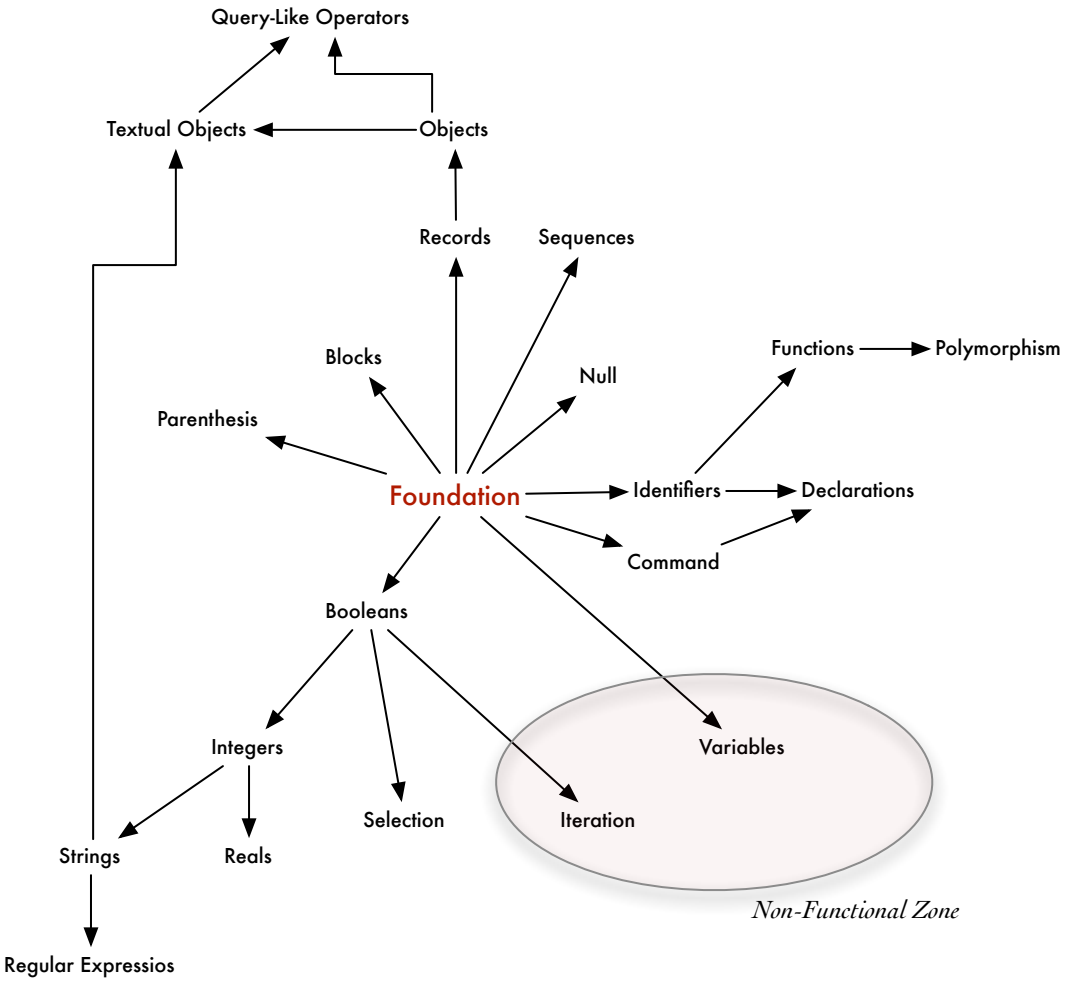

Figure 5.1: The Manuzio dependencies between bundles organized as a graph.

compatibility rules determine when values of a certain type can be used in a given context. Type inference rules define the type of an expression based on the types of its constituents parts.

Type checking is the process of ensuring that a program conforms to the language's type rules. A violation of those rules is called a *type clash*. *Strongly typed* languages, like Manuzio, prevent the execution of programs that do not pass the type check. The application of any operation to any object that is not intended to support that operation is prohibited. A language is said to be *statically typed* if it is strongly typed and type checking can be performed before the actual execution of the program.

The Manuzio language has a small set of predefined types, such as integers or booleans, but allows the construction of other, more complex types, through the use of type constructors. Constructed, or composite, types must be *declared* and *defined* before being used. A declaration introduces a name and indicate, usually implicitly, the scope in which the name will be visible. A definition, instead, describes how a type is constructed by applying a type constructor to a set of already defined types. Types can be thought of in at least three ways, called *denotational*, *constructive*, and *abstraction-based*. From a denotational point of view, types are seen as sets of values. From the constructed point of view, instead, a type is either a primitive, built-in, type or a composite type created by applying a type constructor to one or more simpler types. Finally, from an abstraction-based point of view, a type is an interface consisting of a set of well-defined operations.

In Manuzio, type equivalence is defined as *structural equivalence*. Structural equivalence is based on the content of definitions: two types are the same if they consists of the same components. While most object-oriented languages use *name equivalence*, instead, we found that the particular domain of application of Manuzio makes a structural approach well-suited. The classical objection to structural equivalence, where two types with the same components but very different semantics are compared, does not apply to textual objects, where the semantics is constrained. Manuzio do not feature any implicit *type conversion* mechanism at the current level of prototyping. Explicit type conversions, or *casts*, have been implemented but, since they do not apply to textual objects, their semantics has been left out of this work.

5.3.1 Type Environments

To successfully perform the type-checking of the language sentences we need to store informations about the fundamental types of the language, as well as the type of the identifiers, both type identifiers and value identifiers, that will be declared during a program life. We begin by defining a logical storage for the predefined types of the language.

Definition 17 *A type constants environment γ_0 is a set of type constants called the*

fundamental types of the language.

While in most languages the set of fundamental types is fixed, the modular nature of the Manuzio language makes it a mutable set. Adding new features to the language through the use of bundles can enrich the type constants environment with new types, such as integers, booleans and so on.

In Manuzio it is possible to define new type names using by the use of *declarations*. To allow the type-checking of expressions containing such types it is necessary to include a type constraint system to store type expressions. A type constant is a type identifier which semantics is equivalent to another type identifier or to one of the predefined types of the language.

Definition 18 *A static type identifiers environment is defined as follows:*

- $\gamma = \emptyset$ *is a type identifiers environment.*
- $\gamma' = \gamma \cup \{T : \tau\}$ *is a type identifiers environment.*

we call the relations in the form $\{T : \tau\}$ type identifier constraints.

To store the type of values an identifiers environment is defined as follows.

Definition 19 *An identifiers environment ε is a set of pairs (x, T) composed of a type expression T and an identifier or constant x. The relation $x : T \in \varepsilon$ can be written $\varepsilon(x) = T$.*

The identifiers environment can be seen as partitioned in two distinct environments. One, called εC stores the types of constants introduced by the bundles of the language (like, for instance, that the constant *1* is of type *Integer*), while the other, called εI stores the pairs of value identifiers and their type (like, for instance, x is of type *Boolean*). The union of the type identifiers environment and the expressions type environment is called the static types environment.

5.3.2 Type Equivalence

The Manuzio language type equivalence is a structural equivalence. In structural type equivalence two types are considered compatible if and only if they have the same structure. Since two types with different names can be in effect "the same" type the type system must be able to compute the structure of each type and to compare them.

The *representation type* function $\rho(T)$ returns a type expressions T' obtained by replacing all the occurrences of type identifiers in T with their definitions, recursively, until no more type identifiers are present. If two types are compatible we can write $T \cong T'$.

Definition 20 *Let T be a type of the language, then:*

1. *if T is a fundamental type of the language then $\rho(T) \cong T$.*
2. *if T is a type identifier then $\rho(T) \cong S$ if and only if $\gamma(T) = S$.*
3. *for any other type we must refer to its specific type equivalence rules to determine the value of $\rho(T)$.*

The function $\rho(T)$ is guaranteed to terminate because the first clause rules out recursive or mutually recursive type definitions.

The type equivalence relation in Manuzio is reflexive :

$$\gamma, \varepsilon \vdash T \cong T \qquad \text{(types-reflexive)}$$

and also transitive:

$$\frac{\gamma, \varepsilon \vdash T \cong T', T' \cong T''}{\gamma, \varepsilon \vdash T \cong T''} \qquad \text{(types-transitive)}$$

In Manuzio each free identifier has a type as defined in the type environment:

$$\gamma, \varepsilon \cup \{id : T\} \vdash id : T \qquad \text{(types-identifier)}$$

each constant has a type as defined by the constant values environment:

$$\gamma, \varepsilon \vdash c : C \qquad \text{(types-constant)}$$

where C is the pre-assigned type for the built-in constant c.

5.3.3 Sub-typing Rules

A type T is subtype of another type S if a value of type T can be used in any context where a value of type S is expected. Such a relation can be written $T <: S$, and S is said to be a super-type of T. In the Manuzio programming language a type S is subtype of another type T if and only if their representation types are in such relation:

$$\frac{\gamma, \varepsilon \vdash \rho(S) <: \rho(T)}{\gamma, \varepsilon \vdash S <: T} \qquad \text{(subtype)}$$

the subtype relation is transitive and reflexive:

$$\gamma, \varepsilon \vdash T <: T \qquad \text{(subtype-reflexive)}$$

$$\frac{\gamma, \varepsilon \vdash \rho(S) <: \rho(T), T <: U}{\gamma, \varepsilon \vdash S <: U} \qquad \text{(subtype-transitive)}$$

This is only a generic rule, the sub-typing rules for each specific type will be introduced precisely in the rest of the chapter. The sub-typing definition can be made more concrete by introducing the subsumption rule, a formal way to inform the type checker that a value of type T can actually masquerade a value of type S if and only if $T <: S$.

$$\frac{\gamma, \varepsilon \vdash E : S, S <: T}{\gamma, \varepsilon \vdash E : T} \qquad \text{(subtype-subsumption)}$$

These definitions are the basic type-checking rules of the Manuzio language. The type-specific rules will be given by each of the types that will be added to the language in the next sections.

5.4 Values

In this section we will provide the basics of the Manuzio semantic rules. Values are the results of the evaluation of expressions that are statically been proved valid by the type-checker. Since expressions can include identifiers, a values environment is needed to store the relations between such identifiers and their value.

Definition 21 *A values environment Γ is a set of finite associations between identifiers and values in the form $x = v$ where each x is unique in Γ and v is a value. The relation $x = v \in \Gamma$ can be written $\Gamma(x) = v$. If in a values environment Γ exists the relation $\Gamma(x) = v$ and an expression yields to add the relation $x = v'$ to Γ then $\Gamma \cup \{x = v'\} = \Gamma'$ and $\Gamma'(x) = v'$.*

The basic kind of values are the constants. While traditional programming languages has a finite set of predefined constants, also called literals, Manuzio is a modular language where new bundles can extend the set of constant present in the language.

Definition 22 *A* constant environment Δ *is defined as follows:*

1. *The empty set $\emptyset$ is a constant environment.*

2. *If Delta is a constant environment and δ is a literal with an associated value of $\underline{v}$ of type T then $\Delta' = \Delta \cup \{\delta = \underline{v} : T\}$ is a constant environment.*

The constant environment extension is a clean and simple way to extend the interpreter with new literals. The extension notation includes informations about the characters sequences that identify such literals, their types and their value in the language. For instance, including the `Integer` bundle (see 5.5.5) yields to the following constant environment extension.

$$\Delta' = \Delta \cup \{0 = \underline{0} : Int, 1 = \underline{1} : Int, ...\} \tag{5.11}$$

This notation is used to tell the language that, in the resulting environment Δ', the literal 0 is a constant of type *Int* that must be interpreted as an integer value $\underline{0}$. By convention, in this thesis, we write in plain text the literals and we use an underlined notation to denote semantic values and semantic operations.

The Manuzio programming languages also includes the concept of references and memory. A reference is a value yielded by the evaluation of a location expression, and the memory is a set of locations. Each location can contain a value of any type. Values contained in memory locations can be updated, that is, substituted by other values as the results of expression evaluations.

Definition 23 *A memory M is a finite set of locations such as:*

1. *The empty set $\emptyset$ is a valid memory.*

2. *If M is a valid memory, l is a location and v is a value then $M \cup \{l = v\}$ is a valid memory.*

If $l = v$ is an element of M we can write $(l = v) \in M$ or $M(l) = v$. Each l is unique in M, if in memory M exists the relation $M(l) = v$ and an expression yields to add the relation $l = v'$ then $M \cup \{x = v'\} = M'$ and $M'(x) = v'$. We can conveniently write this behavior as $M \leftarrow \{v' \setminus v\}$.

For a more comprehensive discussion about memory locations, addresses and the effect that the introduction of such constructs has on the language, see 5.5.7.

5.5 Language Elements

In this section the Manuzio core that have been given in the previous sections will be enriched with other language *elements*. An element can be informally defined as a logical portion of the language that adds a meaningful set of functionalities. An example of element is the *Integer* element. In general, by defining an element the following features can be added to the language:

1. New types: one or more new fundamental types and type constructors can be added. For instance, the `Integer` bundle adds the `Int` type to the language.

2. New constants: the constants values environment can be extended to include new values. For instance the `Integer` bundle introduces a syntax to denote integer constants, the `String` bundle introduces string literals, and so on.

3. New values. For instance the `IntValue(0)`, `IntValue(1)`, ... are introduced by the `Integer` bundle.

4. New expressions: to make use of new values, constants and types new expressions are defined. The `Integer` bundle, for instance, introduces the classical arithmetic expressions between integers.

5. Static semantics rules: if a new type is introduced in this section we will find the rules to calculate its free type variables, the substitution rules, the well-formedness rules and the subtyping rules. For each expression the free identifiers and visibility rules are given, along with the type checking rules.

6. Dynamic semantics: here each of the new expressions's semantics is given.

7. Extra: some particular constructs of the language can require additional extensions that will be discussed when presented. For example, when discussing the `TextualObject` bundle, we will have to introduce the concept of *persistent environment*.

If a bundle does not implement one or more of the previous features, then that entry will be omitted in its description. For example, since relational operators will not define any new type nor constant or values, their relative entries will be left out.

5.5.1 Commands

The `Command` bundle introduces one of the fundamental types of the Manuzio Language, the *Command* type. The *Command* type has only one possible value, called *nop*, that is the value of all the expressions that manipulate the environments, such as declarations, variable assignments, while loops and so on. The *Command* type is different from the *NULL* type, as we will see in Section 5.5.8 when the *NULL* type will be introduced. While the behavior of the two types is similar, they denotes very different concepts, the former signal that an expression brings some side-effect, while the latter denotes a non-existent or unknown value.

Note that, since the only value introduced by the bundle, *nop*, is not a denotable value, we can skip the entire dynamic semantics section of the bundle. The Command type and its only value will only be used as yielded values of other expressions, and they have not any semantic use.

Types Environments Extension

The static types environment γ is extended with the new type *Command*, obtaining the new static types environment γ'.

$$\gamma' = \gamma \cup \{Command\} \tag{5.12}$$

Constants Environment Extension

The constants environment Δ not extended, since the only value introduced by the bundle, *nop*, is not intended to be used directly anywhere in the language.

Syntax

The `Command` bundle does not defines any new expression, since the *nop* value is not intended to be used by any expression.

Free Type Variables and Type Substitutions

The Command type is not a type constructor, so it does not yield to any free type variable.

$$FTV(Command) = \emptyset \qquad (\text{command}\binom{ftv}{stat})$$

For the same reason, substitutions do not affect the Command type.

$$Command[X_i \leftarrow \tau_i^{i \in 1..n}] = Command \qquad (\text{command}\binom{subs}{stat})$$

Type Checking Rules

We only need a simple rule to tell the interpreter the type of the *nop* value.

$$\gamma, \varepsilon \vdash nop : Command \qquad \text{(command-}binomexpstat\text{)}$$

Subtyping Rules

The *Command* type is only subtype of itself.

$$\gamma, \varepsilon \vdash Command <: Command \qquad \text{(command-}binomsubstat\text{)}$$

5.5.2 Identifiers

In this section the behavior of type and value identifiers of the Manuzio language is explained. An identifier is a string that will be bound with a type or a value of the language. We will refer to the sequence of characters that form that string as the identifier's name. The name of the identifier must be composed of alphanumeric characters and must start with a letter or an underscore. Moreover, identifier names must be different from the language keywords or unexpected results (most of the time a parsing error) can happen.

A *type identifier* is a name bound to a type expression. Each type identifier can be only declared once, or the interpreter will issue a static error. Type identifiers must begin with an uppercase letter (the convention, however, is to use capitalized identifiers) and can be considered as an alias for primitive types and type constructors instances. A *value identifier* is a name that will become bound to a value. A value identifier must start with a lowercase letter or with an underscore.

Syntax

The syntax of identifiers is trivial, they are denoted by their name. In the following syntax X is a type identifier and x a value identifier.

$$X \qquad \text{(type-ide)}$$

$$x \qquad \text{(value-ide)}$$

Static Semantics

The static semantics rules for dealing with identifiers follow.

Free Variables and Type Substitutions

Since we are defining a new type of the language, the type identifier, we must also define what are it's free type variables. The only free type variable of a type identifier is the type identifier itself.

$$FTV(X) = X \qquad \text{(type-ide}\tbinom{ftv}{stat}\text{)}$$

When we perform a substitutions of the form $[X_i \leftarrow \tau_i]$ with $i \in 1..n$ a type identifier gets substituted if and only if it is present in the list of substitutions.

$$X[X_i \leftarrow \tau_i] = \begin{cases} X & \text{if } X \notin X_i \\ X_i & \text{if } X = X_i \end{cases} \qquad \text{(type-ide}\tbinom{subs}{stat}\text{)}$$

Well-formedness

A type identifier is well-formed only if it has been declared and thus it is present in the type environment.

$$\gamma \cup \{X = \tau\}, \varepsilon \vdash X \qquad (\text{type-ide}\binom{well}{stat})$$

Subtyping Relations

A type identifier is subtype only of itself. Remember that, due to the type equivalency, this means that a type identifier is subtype of all the types that are equivalent to it. We could write:

$$\frac{\gamma, \varepsilon \vdash \rho(X) <: T}{\gamma, \varepsilon \vdash X <: T} \qquad (\text{type-ide}\binom{sub}{stat})$$

but with type equivalency rule we gave earlier we can simply write, without omitting anything:

$$\gamma, \varepsilon \vdash X <: X \qquad (\text{type-ide}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

A value identifier is a free identifier and does not introduces any new type definition.

$$FV(x) = x \qquad (\text{val-ide}\binom{fv}{stat})$$

$$DV(x) = \emptyset \qquad (\text{val-ide}\binom{dv}{stat})$$

Type Checking Rules

The type of a type identifier is the type itself. The type of a value identifier is the type of the value that is associated to it.

$$\gamma \cup \{x = \tau\}, \varepsilon \vdash x : \tau \qquad (\text{val-ide}\binom{exp}{stat})$$

Dynamic Semantics

Evaluating a value identifier yields the value to which that identifier is bound.

$$\Gamma \cup \{x = v\}, M \vdash x \Rightarrow v \qquad (\text{val-ide}\binom{exp}{dyn})$$

5.5.3 Declarations

This section will introduce two fundamental expressions to the Manuzio language, called declarations. A declaration can be of two kinds, a type declaration that introduces a new type identifier or a value declaration that introduces a new value identifier.

Processing a declaration results in adding new bound into an environment. The symbol $\diamond$ is used to enrich the type checking rules of declarations with the effect they have on the environments. Both axioms and rules can be enriched in such a way. We call such rules *judgements*. For example the judgement:

$$\gamma, \varepsilon \vdash \mathit{declaration} : T \diamond \gamma', \varepsilon' \qquad \text{(judgemnt-axiom)}$$

tells that, when type-checked in a type constraint system γ and a static type environment ε, the expression *expression* has type T and it leads to a new type constraint system γ' and a new type environment ε'.

Syntax

A type declaration is introduced by the keyword *type* followed by the name of the new *type identifier*. The *type expression* after the equals can be any valid type, type constructor or type identifier of the language.

$$\textbf{type}\ \tau = T \qquad \text{(type-dec)}$$

$$\textbf{let}\ x = e \qquad \text{(let-dec)}$$

$$\textbf{let}\ x : \tau = e \qquad \text{(let-dec-typed)}$$

Where τ is a type identifier, T is a type expression, x is an identifier and e is any expression. It is worth notice that since Manuzio is an expression-based language, both type and value declarations are just another kind of expressions.

Static Semantics

The following paragraphs describes the static semantics rules for declarations.

Free Identifiers and Visibility Rules

A type declaration does not introduces new free or definition identifiers.

$$FV(\text{type } \tau = T) = \emptyset \qquad \text{(type-dec}\tbinom{fv}{stat}\text{)}$$

$$DV(\text{type } \tau = T) = \emptyset \qquad (\text{type-dec}\binom{dv}{stat})$$

Value declarations, typed or untyped, declare a new identifier and their free identifiers are the free identifiers of the expression that yield the identifiers value.

$$FV(\text{let } x = e) = FV(e) \qquad (\text{let-dec}\binom{fv}{stat})$$

$$DV(\text{let } x = e) = \{x\} \qquad (\text{let-dec}\binom{dv}{stat})$$

$$FV(\text{let } x : \tau = e) = FV(e) \qquad (\text{type-dec-typed}\binom{fv}{stat})$$

$$DV(\text{let } x : \tau = e) = x \qquad (\text{type-dec-typed}\binom{dv}{stat})$$

Type Checking and Sub-typing Rules

A type declaration type is *Command* and checking the declaration extends the type environment with the new type identifier.

$$\gamma, \varepsilon \vdash type\ \tau = T : Command \diamond \gamma \cup \{\tau = T\}, \varepsilon \qquad (\text{type-dec}\binom{exp}{stat})$$

The value declaration type checking rules are dependent on the presence of a type in the left-hand part of the expression. If the type constraint is not present the declaration just yield to a new entry in the type environment. In the other case the type checker ensures that the given type identifier and the type of the right-hand expression are compatible. In both cases the type of the *let* expression is *Command*.

$$\frac{\gamma, \varepsilon \vdash e : T}{\gamma, \varepsilon \vdash let\ x = e : Command \diamond \gamma, \varepsilon \cup \{x : T\}} \qquad (\text{let-dec}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : T,\ T <: S}{\gamma, \varepsilon \vdash let\ x : S = e : Command \diamond \gamma, \varepsilon \cup \{x : S\}} \qquad (\text{let-dec-typed}\binom{exp}{stat})$$

Dynamic Semantics

The dynamic semantics of declarations is straightforward.

$$\Gamma, M \vdash type\ \tau = T \Rightarrow nop \qquad (\text{type-dec}\binom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash let\ x = e \Rightarrow nop \diamond \Gamma \cup \{x = v\}, M} \qquad (\text{let-dec}\binom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash let\ x : T = e \Rightarrow nop \diamond \Gamma \cup \{x = v\}, M} \qquad (\text{let-dec-typed}\binom{exp}{dyn})$$

Examples

Some simple usage of the declaration bundle expressions follows. Since this is one of the first bundle we introduce the examples are trivial. During the rest of the section the language will be enriched by the introduction of new types and expressions and the examples will more and more interesting.

```
1        type T = Int          #=> nop : Command & ET <- {T = Int}
2        let a:T = v           #=> nop : Command & EV <- {a = v}
3        let b = u             #=> nop : Command & EV <- {b = u}
```

In the listing we use the notation described in Chapter 4 where we write the sentence followed by the # => symbol. After that symbol we write the value and type of the sentence and, if the sentence modifies the environment, we append the symbol & and how the environment gets modified.

5.5.4 Booleans

The `Booleans` bundle adds the common primitive *Boolean* type to the Manuzio language. In Manuzio there are two boolean constants, *true* and *false*, with the obvious semantic value.

Types Environment Extension

The static types environment Γ is extended with the *Bool* type.

$$\gamma' = \gamma \cup \{Bool\} \tag{5.13}$$

Constants Environment Extension

The constants environment is extended to add the *true* and *false* literals to the language. both of type *Bool*. Their semantic value is the obvious truth value of true/yes/1/on for *true* and false/no/0/off for *false*.

$$\Delta' = \Delta \cup \{true = \underline{true} : Bool,\ false = \underline{false} : Bool\} \tag{5.14}$$

Syntax

The `Booleans` bundle includes the most basic operations on booleans *and*, *or*, *not*.

$e\ and\ e$ (bool-and)

$e\ or\ e$ (bool-or)

$not\ e$ (bool-not)

Static Semantics

The following paragraphs introduces all the rules the type checker needs to handle boolean values.

Free Type Variables and Type Variables Substitution

Since the *Bool* type is not a type constructor it does not yield to any free type variable.

$$FTV(Bool) = \emptyset \qquad (\text{bool}\tbinom{ftv}{stat})$$

For the same reason, when a type variable substitution is invoked on the *Bool* type, the results, independently of the substitution, is again of type *Bool*.

$$Bool[X_i \leftarrow \tau_i^{i \in 1..n}] = Bool \qquad (\text{bool}\binom{sub}{stat})$$

Well Formedness

The *Bool* type is always well formed.

Subtyping Relations

The *Bool* type is subtype only of itself.

$$\gamma, \varepsilon \vdash Bool <: Bool \qquad (\text{bool}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the boolean operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e\ op\ e') = FV(e) \cup FV(e')\ \forall\ op \in \{and, or\} \qquad (\text{bool-op}\binom{ftv}{stat})$$

$$FV(not\ e) = FV(e) \qquad (\text{bool-not}\binom{ftv}{stat})$$

Type Checking Rules

The type checking rules for booleans are presented in the following formulas.

$$\frac{\gamma, \varepsilon \vdash e : Bool, e' : Bool}{\gamma, \varepsilon \vdash e\ op\ e' : Bool}\ \forall\ op\ \in \{and, or\} \qquad (\text{bool-op}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Bool}{\gamma, \varepsilon \vdash not\ e : Bool} \qquad (\text{bool-not}\binom{exp}{stat})$$

Dynamic Semantics

To specify the dynamic semantics of boolean values we assume to have the logical operations *and*, *or* and *not* as basic semantic operations. We will refer to such operations with {$\underline{and}$, $\underline{or}$, $\underline{not}$}. Given these semantic operators the dynamic semantics rules are straightforward.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e \ op \ e' \Rightarrow v \ \underline{op} \ v'} \ \forall \ op \ \in \{and, or\}, \ \underline{op} \in \{\underline{and}, \ \underline{or}\} \quad (\text{bool-op}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash not \ e \Rightarrow \underline{not} \ v} \quad (\text{bool-not}\tbinom{exp}{dyn})$$

5.5.5 Integers

The `Integers` bundle adds the common integer type to the Manuzio language. This is one of the most common features of any language and it is also one of the primitive types of Manuzio. Many other bundles will depend on `Integer`. The formalism of integers is so common that we will not explain it in depth here, assuming the reader is familiar with it.

Types Environment Extension

The static types environment γ is extended with the new *Int* type, representing the type of all integer values of the language.

$$\gamma' = \gamma \cup \{Int\} \tag{5.15}$$

Constants Environment Extension

The constants environment Δ is extended to include all the denotable natural integer numbers. Each of such constants semantics is their common numerical meaning, so the constant composed by the characters *1* and *0* is translated in the integer value representing the number ten.

$$\Delta' = \Delta \cup \{0 = \underline{0} : Int, 1 = \underline{1} : Int, ...\} \tag{5.16}$$

Syntax

The `Integer` bundle introduces new expressions representing the common integer arithmetic and relational operators.

$$e + e \tag{int-sum}$$

$$e - e \tag{int-sub}$$

$$e * e \tag{int-mult}$$

$$e/e \tag{int-div}$$

$$-e \tag{int-neg}$$

$$e > e' \tag{int-gt}$$

$$e < e' \tag{int-lt}$$

$$e \geq e' \qquad \text{(int-gte)}$$

$$e \leq e' \qquad \text{(int-lte)}$$

$$e = e' \qquad \text{(int-eq)}$$

$$e \; ! = e' \qquad \text{(int-neq)}$$

Static Semantics

The following paragraphs introduces all the rules the type checker needs to handle integer numbers.

Free Type Variables and Type Variables Substitution

Since the *Integer* type is not a type constructor it does not yield to any free type variable.

$$FTV(Int) = \emptyset \qquad (\text{int}\binom{ftv}{stat})$$

For the same reason, when a type variable substitution is invoked on the integer type, the results is again the integer type, independently of the substitution.

$$Int[X_i \leftarrow \tau_i^{i \in 1..n}] = Int \qquad (\text{int}\binom{subs}{stat})$$

Well Formedness

The integer type is always well formed.

$$\gamma, \varepsilon \vdash Int \qquad (\text{int}\binom{well}{stat})$$

Subtyping Relations

The *Int* type is subtype only of itself.

$$\gamma, \varepsilon \vdash Int <: Int \qquad (\text{int}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the integer operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e\ op\ e') = FV(e) \cup FV(e')\ \forall\ op \in \{+, -, *, /, <, >, \leq, \geq, =, !=\} \quad (\text{int-op}\binom{ftv}{stat})$$

$$FV(-e) = FV(e) \quad (\text{int-neg}\binom{ftv}{stat})$$

Type Checking Rules

The type checking rules for integers are presented in the following formulas. The first rule is used when typecheking arithmetic infix operators, while the second is used for relational comparisons. The last rule refer to the unary minus operator.

$$\frac{\gamma, \varepsilon \vdash e : Int, e' : Int}{\gamma, \varepsilon \vdash e\ op\ e' : Int}\ \forall\ op\ \in \{+, -, *, /\} \quad (\text{int-arop}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Int, e' : Int}{\gamma, \varepsilon \vdash e\ op\ e' : Bool}\ \forall\ op\ \in \{<, >, \leq, \geq, =, !=\} \quad (\text{int-relop}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Int}{\gamma, \varepsilon \vdash -e : Int} \quad (\text{int-neg}\binom{exp}{stat})$$

Dynamic Semantics

To specify the dynamic semantics of integer arithmetic expressions we assume to have the four integers basic operations as basic semantic operations. We will refer to such operations with $\{\underline{+}_i,\ \underline{-}_i,\ \underline{*}_i,\ \underline{/}_i\}$ where the $\underline{-}$ operator is used to denote both the unary integer negation and the binary subtraction. In the same way we assume to have basic relational semantic operations such as $\underline{\leq}_i$, $\underline{\geq}_i$, and so on and we use them to express the dynamic semantics of comparisons between integers. Given these semantic operators the dynamic semantics rules are straightforward.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e\ op\ e' \Rightarrow v\ \underline{op}\ v'}\ \forall\ op\ \in \{+, -, *, /\},\ \underline{op} \in \{\underline{+}_i, \underline{-}_i, \underline{*}_i, \underline{/}_i\} \quad (\text{int-arop}\binom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash -e \Rightarrow \underline{-}_i v} \quad (\text{int-neg}\binom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e \ op \ e' \Rightarrow v \ \underline{op} \ v'} \ \forall \, op \ \in \{<, >, \leq, \geq, =, \mathrel{!}=\}, \ \underline{op} \in \{\underline{<_i}, \underline{>_i}, \underline{\leq_i}, \underline{\geq_i}, \underline{=_i}, \underline{\mathrel{!}=_i}\}\}$$

$$(\text{int-relop}\tbinom{exp}{dyn})$$

5.5.6 Reals

The `Reals` bundle add the primitive *Real* type to the Manuzio language to represent floating-point numbers.

Types Environment Extension

The static types environment γ is extended with the new *Real* type, representing the type of all real values of the language.

$$\gamma' = \gamma \cup \{Real\} \tag{5.17}$$

Constants Environment Extension

The constants environment Δ is extended to include all the denotable positive real numbers. In Manuzio a real number is denoted by one or more numeric characters followed by a dot and one or more other numeric characters. Each of such literals semantics is their common numerical meaning, so the constant composed by the characters '1', '.' and '0' is translated in the real value representing the number 1.0. Since reals cannot be enumerated we use a regular expression like notation to formalize the constants environment extension.

$$\Delta' = \Delta \cup \{[0-9]^+.[0-9]^+\} \tag{5.18}$$

Syntax

The `Reals` bundle introduces new expressions representing the common arithmetic operators among real numbers.

$$e + e \tag{real-sum}$$

$$e - e \tag{real-sub}$$

$$e * e \tag{real-mult}$$

$$e/e \tag{real-div}$$

$$-e \tag{real-neg}$$

$$e > e' \tag{real-gt}$$

$$e < e' \tag{real-lt}$$

$$e \geq e' \qquad \text{(real-gte)}$$

$$e \leq e' \qquad \text{(real-lte)}$$

$$e = e' \qquad \text{(real-eq)}$$

$$e \mathrel{!}= e' \qquad \text{(real-neq)}$$

Static Semantics

The following paragraphs introduces all the rules the type checker needs to handle real numbers.

Free Type Variables and Type Variables Substitution

Since the *Real* type is not a type constructor it does not yield to any free type variable.

$$FTV(Real) = \emptyset \qquad (\text{real}\binom{ftv}{stat})$$

For the same reason, when a type variable substitution is invoked on the real type, the results is again the real type, independently of the substitution.

$$Real[X_i \leftarrow \tau_i^{i \in 1..n}] = Real \qquad (\text{real}\binom{subs}{stat})$$

Well Formedness

The real type is always well formed.

$$\gamma, \varepsilon \vdash Real \qquad (\text{real}\binom{well}{stat})$$

Subtyping Relations

The *Real* type is subtype only of itself.

$$\gamma, \varepsilon \vdash Real <: Real \qquad (\text{real}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the real operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e\ op\ e') = FV(e) \cup FV(e')\ \forall\ op \in \{+, -, *, /, <, >, \leq, \geq, =, !=\} \quad (\text{real-op}\tbinom{ftv}{stat})$$

$$FV(-e) = FV(e) \quad (\text{real-neg}\tbinom{ftv}{stat})$$

Type Checking Rules

The type checking rules for reals are presented in the following formulas.

$$\frac{\gamma, \varepsilon \vdash e : Real, e' : Real}{\gamma, \varepsilon \vdash e\ op\ e' : Real}\ \forall\ op\ \in \{+, -, *, /\} \quad (\text{real-arop}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Real, e' : Real}{\gamma, \varepsilon \vdash e\ op\ e' : Bool}\ \forall\ op\ \in \{<, >, \leq, \geq, =, !=\} \quad (\text{real-relop}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Real}{\gamma, \varepsilon \vdash -e : Real} \quad (\text{real-neg}\tbinom{exp}{stat})$$

Dynamic Semantics

Like we did for integers to specify the dynamic semantics of real numbers we assume to have four floating point operations as basic semantic operations. We will refer to such operations with $\{\underline{+_r}, \underline{-_r}, \underline{*_r}, \underline{/_r}\}$ where the $\underline{-_r}$ operator is used to denote both the unary real negation and the binary subtraction. In the same way we assume to have basic relational semantic operations such as $\underline{\leq_r}$, $\underline{\geq_r}$, and so on and we use them to express the dynamic semantics of comparisons between reals. Given these semantic operators the dynamic semantics rules are straightforward.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e\ op\ e' \Rightarrow v\ \underline{op}\ v'}\ \forall\ op\ \in \{+, -, *, /\},\ \underline{op} \in \{\underline{+_r}, \underline{-_r}, \underline{*_r}, \underline{/_r}\} \quad (\text{real-arop}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e\ op\ e' \Rightarrow v\ \underline{op}\ v'}\ \forall\ op\ \in \{<, >, \leq, \leq, =, !=\},\ \underline{op} \in \{\underline{<_r}, \underline{>_r}, \underline{\leq_r}, \underline{\geq_r}, \underline{=_r}, \underline{!=_r}\}$$

$$(\text{real-relop}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash -e \Rightarrow \underline{-_r} v} \qquad (\text{real-neg}\binom{exp}{dyn})$$

5.5.7 Locations and Variables

The `Variables` bundle introduces the classical programming languages concept of variable, a location in a memory, identified by an address, that contains an updateable value. This bundle is based on the concept of memory discussed in Section 5.

By introducing updatable variables in a functional language we are in fact declassing it from a purely functional language to a functional language. A purely functional language has no memory or I/O side effects, (other then the computation of the result). This means that pure functional languages have several useful properties, many of which can be used to optimize the code in a multi processor implementation.

The fact that variables are not a fundamental feature of our modular language, but only a bundle, means that we can easily revert Manuzio to a purely functional language by removing this bundle (possibly along with all other bundles that make it impure).

Memory details like addresses, limits, cell size, and so on are left to the specific implementation, here the concept of memory will be presented in a very simple, high level way. An address is simply an integer, but we will use a notation to distinguish addresses from numbers. The address of the first cell in memory is 0x1 the second one is 0x2 and so on. Cells have a variable size that is always enough to contain the value that they store.

In this section we will explore the new location type and we will present the expressions to create a variable, update it, and access its value.

Types Environment Extension

The `Variables` bundle introduces a new parametric type, or type constructor, in Manuzio, that will be the type of locations, called *vartype*. Each location type must know the type of the value it points to, so we can have many location types like *vartype Int*, *vartype Bool*, and so on. In general the static types environment γ is extended in the following way.

$$\gamma' = \gamma \cup \{vartype\ \tau\}\ \forall\ \tau \in \gamma \tag{5.19}$$

Constants Environment Extension

The `Variables` bundle itself does not define any new constant, but see Section 5.5.8 to learn about the `Null` bundle which defines the *nil* constant, not mandatory but closely tied to the concept of variables.

Syntax

The `Variables` bundle add to the language the expressions needed to declare a variable, to fetch its value from the memory with the *at* operator (!), and to update

its value with the *assignment* operator (:=).

$$var\ e \qquad \text{(var-varvalue)}$$

$$!e \qquad \text{(var-at)}$$

$$e := e \qquad \text{(var-assign)}$$

Static Semantics

The following paragraphs introduces all the rules the type checker needs to handle variable types and expressions involving variables.

Free Type Variables and Type Variables Substitution

Since *vartype* is a type constructor that references another type, the type of the referenced value, it also introduces a free type variable as follows.

$$FTV(vartype\ \tau) = FTV(\tau) \qquad (\text{var}\binom{ftv}{stat})$$

In case of a variable substitution the substitution must be carried over the referenced type.

$$(vartype\ \tau)[X_i \leftarrow \tau_i^{i\in 1..n}] = (vartype\ \tau[X_i \leftarrow \tau_i^{i\in 1..n}]) \qquad (\text{var}\binom{subs}{stat})$$

Well Formedness

The well formedness of a variable type depends directly on the well formedness of the referenced type.

$$\frac{\gamma, \varepsilon \vdash \tau}{\gamma, \varepsilon \vdash vartype\ \tau} \qquad (\text{var}\binom{well}{stat})$$

Subtyping Relations

A variable type τ is subtype of another type τ' if and only if it τ' is a variable type and its referenced type is subtype of the type referenced by τ.

$$\frac{\gamma, \varepsilon \vdash \tau <: \tau'}{\gamma, \varepsilon \vdash vartype\ \tau <: vartype\ \tau'} \qquad (\text{var}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the `Variables` bundle expressions defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(var\ e) = FV(e) \qquad (\text{var-varvalue}\tbinom{exp}{stat})$$

$$FV(!e) = FV(e) \qquad (\text{var-at}\tbinom{exp}{stat})$$

$$FV(e := e') = FV(e) \cup FV(e') \qquad (\text{var-assign}\tbinom{exp}{stat})$$

Type Checking Rules

The *var-varvalue* expression is used to construct variable values, so it must yield to values of type *vartype*.

$$\frac{\Gamma, M \vdash e : T}{\Gamma, M \vdash var\ e : vartype\ T} \qquad (\text{var-varvalue}\tbinom{exp}{stat})$$

The *at* expression is used to fetch a value from the memory, so the type of such expression is the type of the referenced value.

$$\frac{\Gamma, M \vdash e : vartype\ T}{\Gamma, M \vdash !e : T} \qquad (\text{var-at}\tbinom{exp}{stat})$$

Finally, the assignment expression that updates a value is an expression that carry a side effect, so it must be of type *Command*. The left-hand expression type must be variable and the right-hand expression type must be a subtype of the left-hand expression type.

$$\frac{\Gamma, M \vdash e : vartype\ T, e' : S, S <: T}{\Gamma, M \vdash e := e' : Command} \qquad (\text{var-assign}\tbinom{exp}{stat})$$

Dynamic Semantics

To specify the dynamic semantics of variables we only need to refer to the concept of memory discussed in Section 5.4. As a quick reminder if M is a memory we can denote the insertion of a value in a new cell of address l a value v with:

$$M' = M \cup \{l, v\} \qquad (5.20)$$

the storage of a value v in the location l is written:

$$M(l) \leftarrow v \tag{5.21}$$

while the retrieval of a value from a specific address l is written:

$$M(l) = v \tag{5.22}$$

the process of finding a free cell, allocate it, storing the value in a meaningful way, and returning the address of the allocated cell is assumed, and details are left to the actual memory implementation. We only need to declare a function *newloc* that, given a memory M, returns the address of a free cell in M. With these assumptions the dynamic semantics rules are straightforward.

$$\frac{\Gamma, M \vdash e \Rightarrow v, newloc(M) = l}{\Gamma, M \vdash var\ e \Rightarrow l \diamond \Gamma, M \cup \{l, v\}} \quad (\text{var-varvalue}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow l, M(l) = v}{\Gamma, M \vdash !e \Rightarrow v} \quad (\text{var-at}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow l, e' \Rightarrow v}{\Gamma, M \vdash e := e' \Rightarrow nop \diamond \Gamma, M(l) \leftarrow v} \quad (\text{var-assign}\tbinom{exp}{dyn})$$

5.5.8 Null

The `Null` bundle introduces an unknown value called *nil*. The nil value is tied to identifiers to indicate that they have no meaning, do not exists, as well as indicating an uninitialized reference to a location in memory. The *nil* value is the only instance of the *NULL* type, defined to be subtype of all other types of the language. The direct consequence of this fact and the subsumption rule as defined in equation subtype-subsumption is that we enrich all other types of the language with a *nil* value, to represent an unknown. The unknown is meant to be used as a placeholder, an expression can be tested to know if it is *nil* with the *exists* operator but a *nil* expression cannot be evaluated. If such an evaluation is performed a runtime error is raised.

Types Environment Extension

The `Null` bundle introduces a new type called *NULL*.

$$\gamma' = \gamma \cup \{NULL\} \tag{5.23}$$

Constants Environment Extension

The `Null` bundle defines a new constant *nil* of type *NULL* to represent an unknown value. We assume to have a value with the same semantics in the underlying virtual machine used for the implementation, called $\underline{nil}$.

$$\Delta' = \Delta \cup \{nil = \underline{nil} : NULL\} \tag{5.24}$$

Syntax

The bundle introduces, along with the constant *nil*, the *exists* operator that can be used to test a value for being *nil*.

exists e (null-exists)

Static Semantics

The following paragraphs introduces all the rules the type checker needs to handle the *Null* type and expressions involving it.

Free Type Variables and Type Variables Substitution

Since the *NULL* type is not a type constructor it does not yield to any free type variable.

$$FTV(NULL) = \emptyset \qquad (\text{null}\binom{ftv}{stat})$$

For the same reason, when a type variable substitution is invoked on the *NULL* type, the results is again the *NULL* type, independently of the substitution.

$$NULL[X_i \leftarrow \tau_i^{i \in 1..n}] = NULL \qquad (\text{null}\binom{subs}{stat})$$

Well Formedness

The *NULL* type is always well formed.

$$\gamma, \varepsilon \vdash NULL \qquad (\text{null}\binom{well}{stat})$$

Subtyping Relations

As stated in the introduction of the bundle, the *NULL* type is subtype of any other legal type of the language.

$$\frac{\gamma, \varepsilon \vdash \tau}{\gamma, \varepsilon \vdash NULL <: \tau} \qquad (\text{null}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the bundle operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(exists\ e) = FV(e) \qquad (\text{null-exists}\binom{ftv}{stat})$$

Type Checking Rules

The only expression of the bundle, *exists*, yields always a truth value.

$$\frac{\gamma, \varepsilon \vdash e : \tau}{\gamma, \varepsilon \vdash exists\ e : Bool} \qquad (\text{null-exists}\binom{exp}{stat})$$

Dynamic Semantics

To specify the dynamic semantics of the *exists* expression we need to assume a semantic operator η that, applied to a value, returns a truth value true if that value is *nil*, a false otherwise.

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash exists\ e \Rightarrow \begin{cases} true & if\ \eta(v) = \underline{nil} \\ false & otherwise \end{cases}} \quad (\text{null-exists}\tbinom{exp}{dyn})$$

The evaluation of any other operator ρ when applied to of an unknown value yields to a runtime error.

$$\frac{\Gamma, M \vdash e \Rightarrow nil}{\Gamma, M \vdash \rho(e) \Rightarrow \text{ Runtime Error}} \quad (5.25)$$

5.5.9 Functional Abstractions

The concept of procedure, function, and recursive function is common to almost all the recent programming languages. A function is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code. It is coded to obtain a specific set of data values from the calling program (its parameters), and provide a result of a specific type (its return value).

In Manuzio functions and recursive functions are supported and are first order values: they can be passed as a parameter, returned by another function, and so on. The concept of procedure, a functional abstraction that does not returns a value, is not present in Manuzio but can be seen just as a function that returns *nop* without loss of expressive power. Differently from other programming languages, Manuzio uses an explicitly different syntax for functions with or without recursion. Moreover, the identifier used to refer to self in the scope of a recursive function is variable, and gets defined during the function definition itself.

Functional abstractions introduce a new type, the *Function* type to represent the type of functional values. A functional value is also called *closure* or *recursive closure*. The `Function` bundle adds the support for functional abstractions to the Manuzio language, along with the operators to apply a function with a set of actual parameters.

Types Environment Extension

Functional abstractions introduce a new type, the *Function* type. Such type is called *Fun* and is a composite type that stores the types of the function parameters and its return type.

$$\gamma' = \gamma \cup \{Fun(\tau_1, \tau_2, \ldots, \tau_n) : \tau_{n+1}\} \; \forall \; \tau_i \in \gamma \tag{5.26}$$

Syntax

In Manuzio functions and recursive functions are distinguished by a different syntax. Differently from many other languages where a function application is invoked by writing the name of an identifier bound to a function value, in Manuzio there is an explicit operator @ (the *application* operator) to performs such task. Such operator works both for recursive and non-recursive functions.

$$\textbf{fun}(id_1 : \tau_1, id_2 : \tau_2, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \tag{fun}$$

$$\textbf{recfun } id(id_1 : \tau_1, id_2 : \tau_2, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \tag{recfun}$$

$$@e(e_1, e_2, \ldots, e_n) \tag{fun-app}$$

Static Semantics

The static semantics rules for functional abstractions are defined in the following sections.

Free Type Variables and Type Variables Substitution

The free type variables of a *Function* type is the union of the sets of the free type variables of its parameters type and its return type.

$$FTV(Fun(\tau_1, \tau_2, \ldots, \tau_n) : \tau_{n+1}) = \bigcup_{i \in 1..n} FTV(\tau_i) \cup FTV(\tau_{n+1}) \qquad (\text{fun}\binom{ftv}{stat})$$

When substituting types in a functional type it is necessary to perform the substitution on its parameters type and on its return type.

$$(Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1})[X_i \leftarrow \tau_i]^{i \in 1..n} =$$
$$Fun(\tau_1[X_i \leftarrow \tau_i]^{i \in 1..n}, \ldots, \tau_n[X_i \leftarrow \tau_i]^{i \in 1..n}) : \tau_n[X_i \leftarrow \tau_i]^{i \in 1..n} \qquad (\text{fun}\binom{sub}{stat})$$

Well Formedness

A *Function* type is well formed when the type of its parameters and its result type are well formed.

$$\frac{\gamma, \varepsilon \vdash \tau_1, \tau_2, \ldots, \tau_n, \tau_{n+1}}{\gamma, \varepsilon, \vdash Fun(\tau_1, \tau_2, \ldots, \tau_n) : \tau_{n+1}} \qquad (\text{fun}\binom{well}{stat})$$

Subtyping Relations

Manuzio use the classical sub-typing rule for functions: a *Function* type T is subtype of another *Function* type S if and only if the type of S formal parameters are all subtypes of the respective T's formal parameters type and if the return type of T is subtype of the return type of S.

$$\frac{\gamma, \varepsilon \vdash \tau'_1 <: \tau_1, \tau'_2 <: \tau_2, \ldots, \tau'_n <: \tau_n, \tau_{n+1} <: \tau'_{n+1}}{\gamma, \varepsilon, \vdash Fun(\tau_1, \tau_2, \ldots, \tau_n) : \tau_{n+1} <: Fun(\tau'_1, \tau'_2, \ldots, \tau'_n) : \tau'_{n+1}} \qquad (\text{fun}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

To compute the free identifiers of a functional abstraction we must compute the free identifiers of the body and remove from that set the formal parameters identifiers. When dealing with a recursive function the auto-reference identifier must be taken into account too.

$$FV(\textbf{fun}(id_1 : \tau_1, id_2 : \tau_2, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e) = FV(e) - \{id_i\}^{i \in 1 \ldots n} \quad (\text{fun}\binom{fv}{stat})$$

$$FV(\textbf{recfun } id(id_1 : \tau_1, id_2 : \tau_2, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e) = FV(e) - \{id_i\}^{i \in 1 \ldots n} \quad (\text{recfun}\binom{fv}{stat})$$

When using the application expression the free identifiers are the union of the function expression and the actual parameters free identifiers.

$$FV(@e(e_1, e_2, \ldots, e_n)) = (\bigcup_{i \in 1..n} FV(e_1)) \cup FV(e) \qquad (\text{fun-app}\binom{fv}{stat})$$

Type Checking Rules

The type-checking rules for functional abstractions are straightforward. If the body of the function definition has a type compatible with the attended return type in a type environment extended with the bounds between the formal parameters types and identifiers, then the functional abstraction type check is correct and the returned type is a *Function* type.

$$\frac{\gamma, \varepsilon \cup \{id_1 : \tau_1, \ldots, id_n : \tau_n\} \vdash e : \tau_{n+1}}{\gamma, \varepsilon \vdash \textbf{fun}(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e : Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1}} \quad (\text{fun}\binom{exp}{stat})$$

Dealing with recursive functions is similar. We must take into account the self-reference and it's type. The type of the self-reference identifier is the type of the functional abstraction, assuming that it is correct.

$$\frac{\gamma, \varepsilon \cup \{id_1 : \tau_1, \ldots, id_n : \tau_n, id : Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1}\} \vdash e : \tau_{n+1}}{\gamma, \varepsilon \vdash \textbf{recfun } id(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e : Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1}} \quad (\text{recfun}\binom{exp}{stat})$$

A functional abstraction application expression is correct when the expression to which the @ operator is applied is a function and the types of the actual parameters

are compatible with the types of the formal parameters. The yielded type is the return type of the function.

$$\frac{\gamma, \varepsilon \vdash e : Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1}, e_1 : \tau_1, \ldots, e_n : \tau_n}{\gamma, \varepsilon \vdash @e(e_1, \ldots, e_n) : \tau_{n+1}} \quad (\text{fun-app}\binom{exp}{stat})$$

Dynamic Semantics

The value of a function definition is a *closure*. A closure value wraps the function itself with an environment composed by all the free identifiers of the function body and their value at the time of the function definition. In Manuzio functional abstractions obey to what is called static scoping. We can denote a closure as follows, where Γ_{fv} is the free identifiers environment. Recursive closures are analogue for recursive functions.

$$\langle \Gamma_{fv}, \textbf{fun}(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \rangle \tag{5.27}$$

$$\langle \Gamma_{fv}, \textbf{recfun } id(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \rangle \tag{5.28}$$

When a function is defined the interpreter has to compute the closure of that function and yield it as the value of the *fun* or *recfun* expression. If we assume for brevity that:

$$f = \textbf{fun}(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \tag{5.29}$$

$$f_{rec} = \textbf{recfun } id(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e \tag{5.30}$$

the dynamic semantics of functions follows.

$$\frac{\Gamma_{fv} = \{x_i = \Gamma(x_i)\}^{i \in 1 \ldots n} \; \forall \; x \in FV(f)}{\Gamma, M \vdash f \Rightarrow \langle \Gamma_{fv}, f \rangle} \quad (\text{fun}\binom{exp}{dyn})$$

$$\frac{\Gamma_{fv} = \{x_i = \Gamma(x_i)\}^{i \in 1 \ldots n} \; \forall \; x \in FV(f_{rec})}{\Gamma, M \vdash f_{rec} \Rightarrow \langle \Gamma_{fv}, f_{rec} \rangle} \quad (\text{recfun}\binom{exp}{dyn})$$

For function applications the body of the function must be evaluated in an environment composed by the function closure extended with the actual parameters.

$$\frac{\Gamma, M \vdash f \Rightarrow \langle \Gamma_{fv}, [\mathbf{rec}]\mathbf{fun}(id_1 : \tau_1, \ldots, id_n : \tau_n) : \tau_{n+1} \textbf{ is } e\rangle \quad \Gamma, M \vdash e_1 \Rightarrow v_1, \ldots, e_n \Rightarrow v_n \quad \Gamma \cup \{id_1 = v_1, \ldots, id_n = v_n\}, M \vdash e \Rightarrow v}{\Gamma, M \vdash @f(e_1, \ldots, e_n) \Rightarrow v} \quad (\text{fun-app}\binom{exp}{dyn})$$

5.5.10 Blocks

In Manuzio a block is a sequence of valid language's sentences grouped to form a single instruction. Such sentences are enclosed in by the old fashioned *begin ... end* keywords and are separated by a semicolon (a semicolon after the last sentence is optional). The type of a block is the type of the last sentence of the block. For this reason blocks are meant to be used in conjunction with expressions that brings a side effect such as the ones given in Section 5.5.7. Blocks are introduced by the `blocks` bundle defined in this section. The `blocks` bundle is rather simple and does not require much comments.

Syntax

The `blocks` bundle defines only the block statement.

begin $exp;\ exp; \ldots exp[;]$ **end** (block)

Static Semantics

The definitions needed to check the static correctness of blocks follow.

Free Identifiers and Visibility Rules

The free variables of a block can be defined as the union of all the free variables of all its composing sentences.

$$FV(begin\ e_1;\ e_2;\ \ldots\ e_n\ end) = \bigcup_{i=1\ldots n} FV(e_i) \qquad (\text{block}\binom{fv}{stat})$$

Type Checking Rules

The type of a block is the type of its last sentence, provided that all the sentences of the block type-checked correctly.

$$\frac{\gamma, \varepsilon \vdash e_1 : \tau_1, e_2 : \tau_2, \ldots, e_n : \tau_n}{\gamma, \varepsilon \vdash begin\ e_1;\ e_2; \ldots e_n;\ end : \tau_n} \qquad (\text{block}\binom{exp}{stat})$$

Dynamic Semantics

When a block is evaluated all its sentences are evaluated sequentially in appearing order. The value of a block is the value of its last sentence.

$$\frac{\Gamma, M \vdash e_1 \Rightarrow v_1, e_2 \Rightarrow v_2, \ldots, e_n \Rightarrow v_n}{\Gamma, M \vdash begin\ e_1; e_2; \ldots, e_n\ end \Rightarrow v_n} \qquad (\text{block}\binom{exp}{dyn})$$

5.5.11 Parenthesis

The Manuzio language supports the use of parenthesis to override the normal operator precedence rules. Parenthesis are defined by the `Parenthesis` bundle and their use is intuitive.

Syntax

The `Parenthesis` bundle defines only the parenthesis statement.

$$(e) \qquad \text{(par)}$$

Static Semantics

The definitions needed to check the static correctness of parenthesis follow.

Free Identifiers and Visibility Rules

The free variables of a parenthesis expression are simply the free variables of the enclosed expression.

$$FV((e)) = FV(e) \qquad (\text{par}\tbinom{fv}{stat})$$

Type Checking Rules

The type of a parenthesis expression is simply the type of the enclosed expression.

$$\frac{\gamma, \varepsilon \vdash e : \tau}{\gamma, \varepsilon \vdash (e) : \tau} \qquad (\text{par}\tbinom{exp}{stat})$$

Dynamic Semantics

A parenthesis expression gets evaluated by evaluating the enclosed expression and yielding that value.

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash (e) \Rightarrow v} \qquad (\text{par}\tbinom{exp}{dyn})$$

5.5.12 Iteration

Since some of the Manuzio bundles can include imperative-style features in the language, like variables, the `While` bundle adds an imperative iteration statement to the language to make it capable of express imperative-style programs. Of course this is not the intended paradigm of Manuzio, but since the language is modular we can add or remove this feature by just including or not the `While` bundle when interpreting. A *while* statement always yield a *nop* value to indicate the presence of side effects.

Syntax

The `while` bundle defines only the while statement.

$$\textbf{while } \textit{condition} \textbf{ do } e_1; e_2; \ldots; e_n \textbf{ end} \qquad \text{(while)}$$

Static Semantics

The definitions needed to check the static correctness of the while statement follow.

Free Identifiers and Visibility Rules

The free variables of a while statement can be defined as the union of all the free variables of all its composing block and its condition.

$$FV(\textbf{while } e \textbf{ do } e_1; e_2; \ldots; e_n \textbf{ end}) = \bigcup_{i=1\ldots n} FV(e_i) \cup FV(e) \qquad (\text{while}\tbinom{fv}{stat})$$

Type Checking Rules

The type of a while statement is always *Command*, provided that the condition's type is *Bool* and all the enclosed instructions type-check correctly.

$$\frac{\gamma, \varepsilon \vdash e_1 : \tau_1, e_2 : \tau_2, \ldots, e_n : \tau_n, e : \mathit{Bool}}{\gamma, \varepsilon \vdash \textbf{while } e \textbf{ do } e_1; e_2; \ldots; e_n \textbf{ end} : \mathit{Command}} \qquad (\text{while}\tbinom{exp}{stat})$$

Dynamic Semantics

A *while* statement is composed by a *condition* and a *block* of instructions. When evaluating the statement the condition is checked first. The choice of the rule used to evaluate it depends on the condition truth value. If it is *true* then the block is evaluated and then the iteration is repeated, else the block is not evaluated and the *nop* value is returned.

$$\frac{\Gamma, M \vdash e \Rightarrow false}{\Gamma, M \vdash \textbf{while}\ e\ \textbf{do}\ e_1; e_2; \ldots; e_n\ \textbf{end} \Rightarrow nop} \quad (\text{while-false}\binom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow true, e_1 \Rightarrow v_1, e_2 \Rightarrow v_2, \ldots, e_n \Rightarrow v_n}{\Gamma, M \vdash \textbf{while}\ e\ \textbf{do}\ e_1; e_2; \ldots; e_n\ \textbf{end} \Rightarrow \textbf{while}\ e\ \textbf{do}\ e_1; e_2; \ldots; e_n\ \textbf{end}} \quad (\text{while-true}\binom{exp}{dyn})$$

Examples

In the following example a *succ* function is defined to compute the successor of an integer. Then a variable x of *Int* type is initialized to zero and increased with the *succ* function application until it gets to 100. The last instruction test our code by confronting the final value contained in x with the constant integer 99. The result of such instruction is a boolean truth value of true.

```
1    let succ = fun(n:Int):Int is n+1     #=> nop:Command
2    let x = var 0                        #=> nop:Command
3    while ((!x)<100) do
4        x := @succ(!x);
5    end                                  #=> nop:Command
6    !x > 99                              #=> true:Bool
```

5.5.13 Selection

The `Selection` bundle import in the language the fundamental *if ... then ... else ... end* statement. The usage of such statement should be enough familiar to allow us to dive into the syntax and semantics without further explanations.

Syntax

The `if` bundle defines only the if statement, which syntax is the following.

$$\textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end} \tag{if}$$

Static Semantics

The definitions needed to check the static correctness of the if statement follow.

Free Identifiers and Visibility Rules

The free identifiers of a selection are the union of the free identifiers of the condition with the ones on both branches.

$$FV(\textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end}) = FV(e) \cup FV(e_t) \cup FV(e_f) \qquad (\text{if}\tbinom{fv}{stat})$$

Type Checking Rules

Since the check on the condition truth value can only be made at runtime the selection statement needs to enforce type compatibility between its if branch and its else branch. Remember that by type compatibility between two types T and S we mean that either T is subtype of S or vice-versa. The type of the if statement will then be the most general type between its if and else branch types. By doing so we ensure that the type checker is sound independently from the branch chosen at runtime.

$$\frac{\gamma, \varepsilon \vdash e : Bool, e_1 : \tau_1, e_2 : \tau_2, \tau_1 <: \tau_2}{\gamma, \varepsilon \vdash \textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end} : \tau_2} \qquad (\text{if-1}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : Bool, e_1 : \tau_1, e_2 : \tau_2, \tau_2 <: \tau_1}{\gamma, \varepsilon \vdash \textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end} : \tau_1} \qquad (\text{if-2}\tbinom{exp}{stat})$$

Dynamic Semantics

The evaluation of the selection statement is straightforward. First the condition is evaluated, if its value is *true* then the if branch is evaluated and its results is yielded, if it's *false* then the else branch is evaluated and its results is yielded.

$$\frac{\Gamma, M \vdash e \Rightarrow true, e_t \Rightarrow v_t}{\Gamma, M \vdash \textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end} \Rightarrow v_t} \quad (\text{if-true}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow true, e_e \Rightarrow v_e}{\Gamma, M \vdash \textbf{if } e \textbf{ then } e_t \textbf{ else } e_e \textbf{ end} \Rightarrow v_e} \quad (\text{if-false}\tbinom{exp}{dyn})$$

Examples

In the following example we must make use of records, described in Section 5.5.16 to have two simple types that are one subtype of the other.

```
1  if (true) then {a=1} else {a=2, b=3} end    #=> {a=1} : {a:Int}
2  if (false) then {a=1} else {a=2, b=3} end   #=> {a=2, b=3} : {a:Int}
```

The first two lines report the same statement with a different condition. We can see that, in both cases, the yielded value is of type $a : Int$, the most general type between the two branches types. This example shows another behavior of the Manuzio interpreter. The second selection returns a value $a = 2, b = 3$ but it's type is $a : Int$. The static type environment masquerade the presence of the b field to the user, so that that field is unaccessible. Every expression that tried to access it will clash with the type-checker and will not be considered correct. This is an example of how a statically typed language can refuse correct programs, while a dynamically typed language would not have had such a problem. An exemplification of such behavior follows; trying to access the b field cause a semantic error from the interpreter.

```
1  let result = if (false) then {a=1} else {a=2, b=3} end  #=> nop:Command
2  #result is {a=2, b=3} : {a:Int}
3
4  result.a        #=> 2 : Int
5  result.b        #=> Errors::SemanticError, b is not a field of {a:Int}
```

5.5.14 Strings

The Manuzio programming language is aimed to work with text and textual objects, so string handling is of fundamental importance. The `Strings` bundle defines a new type, *String*, that represent a sequence of characters encoded in UTF-8. Each value of that type is called a *string* and can be manipulated through a rich set of operators. In this section we will introduce the new type and values and a collection of such operators. The *concat* operator concatenates two strings, the times operator repeat a string n times, the *size of* operator returns the number of characters in the string. The *slice* operator, instead, is of central importance and allows the selection of substrings. In Section 5.5.15 the regular expressions will be introduced and the `Strings` bundle will be enriched with a pattern matching operator.

Types Environment Extension

The `Strings` bundle introduces a new type, *String*.

$$\gamma' = \gamma \cup String \tag{5.31}$$

Constants Environment Extension

The `String` bundle introduces new constants called *string literals*. A string literal is defined as "any sequence of zero or more characters enclosed in double quotes". From now on we denote with "s" a Manuzio string, while with "s" its semantic value. In the following definition a regular expression is used to denote the concept of "any sequence of zero or more characters".

$$\Delta' = \Delta \cup \{\text{“}s'' = \underline{\text{“}s''} : String\} \ \forall \ s \in /.*/ \tag{5.32}$$

Syntax

The `String` bundle adds the following expressions to the language.

$e + e$ (str-concat)

$e * e$ (str-times)

size of e (str-size)

$e[e'..e'']$ (str-slice)

Static Semantics

Free Type Variables and Type Variables Substitution

Since the *String* type is not a type constructor it does not yield to any free type variable.

$$FTV(String) = \emptyset \qquad (\text{str}\binom{ftv}{stat})$$

For the same reason, when a type variable substitution is invoked on the string type, the results is again the string type, independently of the substitution.

$$String[X_i \leftarrow \tau_i^{i \in 1..n}] = String \qquad (\text{str}\binom{subs}{stat})$$

Well Formedness

The integer type is always well formed.

$$\gamma, \varepsilon \vdash String \qquad (\text{str}\binom{well}{stat})$$

Subtyping Relations

The *String* type is subtype only of itself.

$$\gamma, \varepsilon \vdash String <: String \qquad (\text{str}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the string operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e + e') = FV(e) \cup FV(e') \qquad (\text{str-concat}\binom{ftv}{stat})$$

$$FV(e * e') = FV(e) \cup FV(e') \qquad (\text{str-times}\binom{ftv}{stat})$$

$$FV(\textbf{size of } e) = FV(e) \qquad (\text{str-size}\binom{ftv}{stat})$$

$$FV(e[e'..e'']) = FV(e) \cup FV(e') \cup FV(e'') \qquad (\text{str-slice}\binom{ftv}{stat})$$

Type Checking Rules

The type-checking rules of string expressions are simple.

$$\frac{\gamma, \varepsilon \vdash e : String, e' : String}{\gamma, \varepsilon \vdash e + e' : String} \quad (\text{str-concat}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : String, e' : Int}{\gamma, \varepsilon \vdash e * e' : String} \quad (\text{str-times}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : String}{\gamma, \varepsilon \vdash size\ of\ e : Int} \quad (\text{str-size}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : String, e' : Int, e'' : Int}{\gamma, \varepsilon \vdash e[e'..e''] : String} \quad (\text{str-slice}\tbinom{exp}{stat})$$

Dynamic Semantics

Since the string data type is more complex then other type we already seen in previous sections, like integers or booleans, its dynamic semantics will be slightly more complicated. We assume to have the following semantic operators:

- $\underline{strcat}(s_1, \ldots, s_n)$: performs the concatenation of two or more strings. The results of such operation is a new string composed by all the characters of s_1 followed by all the characters of s_2, and so on.
- $\underline{strlen(s)}$: returns the number of characters composing the string s as a natural number.

Given those semantic operators we can define the dynamic semantics of the `Strings` bundle expressions as follows.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e + e' \Rightarrow \underline{strcat}(v, v')} \quad (\text{str-concat}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e * e' \Rightarrow \underline{strcat}(v, v)^{v'}} \quad (\text{str-times}\tbinom{exp}{dyn})$$

where by $\underline{strcat}(v, v)^{v'}$ we mean the iteration of the operation $\underline{strcat}(v, v)$ for v' times. To iterate zero times yields to the empty string.

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash \textbf{size of}\ e \Rightarrow \underline{strlen}(v)} \quad (\text{str-size}\tbinom{exp}{dyn})$$

The *slice* operator has the classical substring semantics.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v', e'' \Rightarrow v''}{\Gamma, M \vdash e[e'..e''] \Rightarrow v[v'..v'']} \quad (\text{str-slice}\tbinom{exp}{dyn})$$

5.5.15 Regular Expressions

One of the most important feature needed in Manuzio is string manipulation. When working with textual objects we will explore the logic structure of a text in a clean and intuitive way but, sooner or later, we will have to extract the text of one or more of such objects to read it, analyze it, or compare it with some other text. A *regular expression* introduces in Manuzio a concise and flexible way to identify strings of interest by comparing them with a pattern of characters. Regular expressions are written in a formal language described in this section and are processed by the interpreter when the *match* or *extended match* operator is called. Regular expressions and their operators are bundled in the `Regexp` bundle.

Types Environment Extension

The `Regexp` bundle extends the type environment with a new type, *Regexp* that represents a string written in the Manuzio regular expression's formal language.

$$\gamma' = \gamma \cup \{Regexp\} \tag{5.33}$$

Constants Environment Extension

A regular expression constant is denoted by a string of characters enclosed between two \$ symbols. Such characters can be any combination of valid regular expression's characters. Each of such constants have a semantic counterpart that will be used by the underlying virtual machine to perform the matching.

$$\Delta' = \Delta \cup \{\$.*\$ = \underline{\$.*\$} : Regexp\} \tag{5.34}$$

Here we use the notation $.*$ to indicate a sequence of zero or more characters.

Regular Expressions in Manuzio

Describe the syntax and meaning of regular expressions here (similar to the ruby ones, without modifiers).

Syntax

Manuzio supports two operators to deal with regular expressions. The simpler one is the *match* operator, used to know if a string conforms to a regular expression or not. The second one is more advanced and returns a record containing more informations about the eventual matching. See Section 5.5.15 later in this section for more information.

$$e \sim e' \qquad \text{(reg-match)}$$

$$e \wr e' \qquad \text{(reg-ext)}$$

Static Semantics

Free Type Variables and Type Variables Substitution

The *regexp* type does not yield to any free type variable. For the same reason, substitutions of type identifiers have no effect on such type.

$$FVT(Regexp) = \{\} \qquad (\text{reg}\binom{ftv}{stat})$$

Well Formedness

A *regexp* type is always well formed.

$$\gamma, \varepsilon \vdash Regexp \qquad (\text{reg}\binom{well}{stat})$$

Subtyping Relations

The *Regexp* type is subtype only of itself.

$$\gamma, \varepsilon \vdash Regexp <: Regexp \qquad (\text{reg}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the regular expression operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e \ \sim e') = FV(e) \cup FV(e') \qquad (\text{reg-match}\binom{fv}{stat})$$

$$FV(e \wr e') = FV(e) \cup FV(e') \qquad (\text{reg-ext}\binom{fv}{stat})$$

Type Checking Rules

The *match* operator takes a string and a regular expression and returns a boolean.

$$\frac{\gamma, \varepsilon \vdash e : String, e' : Regexp}{\gamma, \varepsilon \vdash e \sim e' : Bool} \quad (\text{reg-match}\tbinom{exp}{stat})$$

The *extended match* operator instead returns a record type containing all the information about the eventual match.

$$\frac{\gamma, \varepsilon \vdash e : String, e' : Regexp}{\gamma, \varepsilon \vdash e \sim e' : \{match : Bool, n : Int, \ldots\}} \quad (\text{reg-ext}\tbinom{exp}{stat})$$

Dynamic Semantics

To specify the dynamic semantics of regular expressions we must assume a semantic operation of pattern matching $\underline{\xi}$. Such operation returns a boolean value of true if a match exists, false if not.

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v', \xi(v, v') = m}{\Gamma, M \vdash e \sim e' = m.match} \quad (\text{reg-match}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v', \xi(v, v') = m}{\Gamma, M \vdash e \sim e' = \{match = m.match, n = m.n, \ldots\}} \quad (\text{reg-ext}\tbinom{exp}{dyn})$$

5.5.16 Records

The `Records` bundle adds record support to the Manuzio language. A *record*, also called *tuple* or *struct*, consists in an aggregation of zero or more values called the record *fields*. Each field can be accessed through an identifier to which it is bound, called *label*. A *record type* is a data type that describes such values. The definition of a record type includes the data type and label of each field. In Manuzio records are of particular importance. As we will see in the following sections they will be used in the definition of both objects and textual objects.

Types Environment Extension

The `Records` bundle adds the *Record* type to the language. Such type is a type constructor that takes in input zero or more labels and the same number of other types. To denote a record type we enclose in curly brackets a list of identifiers, each followed by a colon and its type. In Manuzio an empty record is a legal value and it is denoted by just a couple of curly brackets. In the rest of the section we use underlined curly brackets when they are not part of the syntax to avoid confusion in the notation.

$$\gamma' = \gamma \cup \underline{\{}\ \{id_1 : \tau_1, \ldots, id_n : \tau_n\}, \{\}\ \underline{\}} \qquad \forall(\tau_i \in \gamma,\ id_i \in \iota)^{i \in 1 \ldots n} \tag{5.35}$$

Syntax

A record expression is composed by a couple of curly brackets that enclose a set of zero or more field expressions. A field expression is an identifier, called *label*, followed by and optional type (separated by a colon) and a value (separated by an equal).

$$\{id_1 : \tau_1 = e_1, \ldots, id_n : \tau_n = e_n\} \tag{rec}$$

The basic record operation is the *dot* operation, used to access a specific record field by its label. Moreover the `Records` bundle includes the *extend* and *project* expressions on records. The former takes two records and merge their fields to create a third one, while the latter takes a record and a subset of it's fields and returns a new record with only those selected fields.

$$e.id \tag{rec-dot}$$

$$e\ \textbf{extend}\ e' \tag{rec-ext}$$

$$e\ \textbf{project}\ \{id_1 : \tau_1, \ldots, id_n : \tau_n\} \tag{rec-proj}$$

Static Semantics

Free Type Variables and Type Variables Substitution

A record type free type variables is the union of the free type variables of its fields type.

$$FTV(\{id_1 : \tau_1, \ldots, id_n : \tau_n\}) = FTV(\tau_1) \cup \ldots \cup FTV(\tau_n) \qquad (\text{rec}\binom{ftv}{stat})$$

Well Formedness

A record type is well formed when:

- the identifiers of its fields are valid;
- each identifier is unique;
- the types of its fields is well formed.

$$\frac{\forall i \in (1..n)\ id_i \notin (\{id_1, \ldots, id_n\} - \{id_i\}), id_i \in \iota, \tau_i}{\gamma, \varepsilon \vdash \{id_1 : \tau_1, \ldots, id_n : \tau_n\}} \qquad (\text{rec}\binom{well}{stat})$$

Subtyping Relations

A record R is subtype of another record T when the all the fields of T are also present in S and their type is a subtype of the corresponding type in S.

$$\frac{\gamma, \varepsilon \vdash \tau_1 <: \tau'_1, \ldots, \tau_n <: \tau'_n}{\gamma, \varepsilon \vdash \{id_1 : \tau'_1, \ldots, id_n : \tau'_n, id_{n+1} : \tau'_{n+1}, \ldots id_m : \tau'_m\} <: \{id_1 : \tau_1, \ldots, id_n : \tau_n\}} \qquad (\text{rec}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

$$FV(\{id_1[: \tau_1] = e_1, \ldots, id_n[: \tau_n] = e_n\}) = FV(e_1) \cup \ldots \cup FV(e_n) \qquad (\text{rec}\binom{fv}{stat})$$

$$FV(e.id) = FV(e) \qquad (\text{rec-dot}\binom{fv}{stat})$$

$$FV(e \textbf{ extend } e') = FV(e) \cup FV(e') \qquad (\text{rec-ext}\binom{fv}{stat})$$

$$FV(e \textbf{ project } \{id_1 : \tau_1, \ldots, id_n : \tau_n\}) = FV(e) \qquad (\text{rec-proj}\binom{fv}{stat})$$

Type Checking Rules

The record *dot* expression type checking rule is straightforward, the resulting type is the type of the corresponding identifier, if a field with that identifier exists.

$$\frac{\gamma, \varepsilon \vdash e : \{id_1 : \tau_1, \ldots, id_n : \tau_n\}, id \in id_i^{i \in 1..n}}{\gamma, \varepsilon \vdash e.id : \tau_i} \quad (\text{rec-dot}\tbinom{exp}{stat})$$

The extension operator type check correctly when both operands are records and there no identifiers in common between them.

$$\frac{\begin{array}{r}\gamma, \varepsilon \vdash e : \{id_1 : \tau_1, \ldots, id_n : \tau_n\}, \\ e' : \{id_{n+1}[: \tau_{n+1}] = e_{n+1}, \ldots, id_m[: \tau_m] = e_m\}, \\ id_i \notin \{id_1, \ldots, id_n\} \ \forall \ i \in (n+1)..m\end{array}}{\gamma, \varepsilon \vdash e \ \textbf{extend} \ e' : \{id_1[: \tau_1] = e_1, \ldots, id_m[: \tau_m] = e_m\}} \quad (\text{rec-ext}\tbinom{exp}{stat})$$

The projection operator between a record R and a record type T type checks correctly if T is a record type where each field f_t have a correspondent identifier f_r in R and $\tau_{f_r} <: \tau_{f_t}$.

$$\frac{\begin{array}{r}\gamma, \varepsilon \vdash e : \{id_1 : \tau_1, \ldots, id_n : \tau_n\}, \\ \forall \ j \in (n+1) \ldots m \ \exists i \in (1 \ldots n) : id_i = id_j, \tau_i <: \tau_j\end{array}}{\gamma, \varepsilon \vdash e \ \textbf{project} \ \{id_{n+1} : \tau_{n+1}, \ldots, id_m : \tau_m\} : \{id_{n+1} : \tau_{n+1}, \ldots, id_m : \tau_m\}} \quad (\text{rec-proj}\tbinom{exp}{stat})$$

Dynamic Semantics

A record value is intended to represent a finite map from labels to values where the labels are unique and the values can have different types. The empty record, represented in Manuzio by a pair of curly brackets, is a valid value and its type is the empty record type.

In our abstract semantic machine record values are represented by the classical notion of tuple, denoted by a collection of labels and values enclosed in underlined curly brackets. The empty record is represented by a pair of underlined curly brackets.

$$\frac{\Gamma, M \vdash e_i \Rightarrow v_i^{i \in 1..n}}{\Gamma, M \vdash \{id_1 : \tau_1 = e_1, \ldots, id_n : \tau_n = e_n\} \Rightarrow \underline{\{}id_1 = v_1, \ldots, id_n = v_n\underline{\}}} \quad (\text{rec}\tbinom{exp}{dyn})$$

The dynamic semantics of dot operator is simple and consists in the evaluation of its fields values.

$$\frac{\Gamma, M \vdash e \Rightarrow \{id_1 = v_1, \ldots, id_n = v_n\}, \exists i \in 1 \ldots n : id = id_i}{\Gamma, M \vdash e.id \Rightarrow v_i} \quad (\text{rec-dot}\tbinom{exp}{dyn})$$

The extension of a record is performed by constructing a new tuple that includes the fields of both arguments's tuples.

$$\frac{\begin{array}{c}\Gamma, M \vdash e \Rightarrow \{id_1 = v_1, \ldots, id_n = v_n\}, \\ e' \Rightarrow \{id_{n+1} = v_{n+1}, \ldots, id_m = e_m\}\end{array}}{\Gamma, M \vdash e \textbf{ extend } e' \Rightarrow \{id_1 = v_1, \ldots, id_m = v_m\}} \quad (\text{rec-ext}\tbinom{exp}{dyn})$$

The projection is performed by constructing a new tuple from the argument's one with only the fields specified in the projection.

$$\frac{\begin{array}{c}\Gamma, M \vdash e \Rightarrow \{id_1 = v_1, \ldots, id_n = v_n\}, \\ \forall\ j \in (n+1) \ldots m\ \exists i \in (1 \ldots n) : id_i = id_j\end{array}}{\Gamma, M \vdash e \textbf{ project } \{id_{n+1} : \tau_{n+1}, \ldots, id_m : \tau_m\} \Rightarrow \{id_{n+1} = v_{n+1}, \ldots, id_m = v_m\}} \quad (\text{rec-proj}\tbinom{exp}{dym})$$

5.5.17 Sequences

A sequence is an homogeneous collection of values. The *Sequence* type, introduced by the `Sequences` bundle, is a fundamental concept in Manuzio and in functional programming in general, since such data structure works very well with recursive algorithms. In Manuzio every sequence have a type, including the empty sequence. For this reason, when denoting an empty sequence, a special syntax that require to explicitly specify its type is required. A collection of operators can manipulate sequences by extracting the head or the tail of a sequence, concatenate two sequences, or test for a sequence to have some properties such as containing a specific value.

Types Environment Extension

The `Sequences` bundle extends the types environment with the *Sequence* type. A sequence type is a type constructor that takes another type as a parameter and constructs a sequence of elements of that type.

$$\gamma' = \gamma \cup \{[\tau]\} \ \forall \ \tau \in \gamma \tag{5.36}$$

Note that τ can be any type, even a sequence, so a sequence of sequences is legal, as long as the well formedness rules are verified.

Syntax

A sequence constant can be denoted by a comma separated collection of one or more values enclosed in square brackets. Note that, since we need to know the type of such values, an empty sequence denoted by just a pair of square brackets is not a legal constant since we can not infer the type of its elements. An expression called *emptyseq* will be used to deal with such situation.

$$[e_1, \ldots, e_n] \tag{seq}$$

The syntax of the sequence related expressions follows.

head e (seq-head)

tail e (seq-tail)

e **cons** e' (seq-cons)

isempty e (seq-isempty)

e **append** e' (seq-append)

e **intersect** e' (seq-intersect)

e **union** e' (seq-union)

e **difference** e' (seq-diff)

e **isin** e' (seq-isin)

flatten e (seq-flat)

emptyseq of τ (seq-emptyseq)

collect $|id|$ e **from** e' (seq-collect)

Static Semantics

Free Type Variables and Type Variables Substitution

The free type variables of a sequence type are the free type variables of its elements type.

$$FTV([\tau]) = FTV(\tau) \qquad (\text{seq}\binom{ftv}{stat})$$

Well Formedness

Since all the values of a sequence must be homogeneous they all must have a compatible type. The type of the resulting sequence will be a sequence type which elements have the type of the most general type among its values type.

$$\frac{\gamma, \varepsilon \vdash e_i : \tau_i \ \forall \ i \in 1 \ldots n \qquad \gamma, \varepsilon \vdash \tau_s = \tau_1, \forall i \in 2 \ldots n \tau_s = \begin{cases} \tau_s & \text{if } \tau_i <: \tau_s \\ \tau_i & \text{if } \tau_s <: \tau_i \end{cases}}{\gamma, \varepsilon \vdash [e_1, \ldots, e_n] : [\tau_s]} \qquad (\text{seq}\binom{well}{stat})$$

Subtyping Relations

A sequence S is subtype of another sequence S' if the elements type of S is subtype of the elements type of S'.

$$\frac{\gamma, \varepsilon \vdash \tau <: \tau'}{\gamma, \varepsilon \vdash [\tau] <: [\tau']} \quad (\text{seq}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

Since none of the integer operations defines new identifiers we omit such rules from the list below. The result of such rules is always the empty set.

$$FV(e \ \mathbf{op} \ e') = FV(e) \cup FV(e') \quad (\text{seq-infix}\binom{fv}{stat})$$

$$FV(\mathbf{op} \ e) = FV(e) \quad (\text{seq-prefix}\binom{fv}{stat})$$

$$FV(\mathbf{emptyseqof} \ \tau) = \emptyset \quad (\text{seq-emptyseq}\binom{fv}{stat})$$

$$FV([e_1, \ldots, e_n]) = \bigcup_{i=1\ldots n} FV(e_i) \quad (\text{seq}\binom{fv}{stat})$$

Type Checking Rules

The typechecking rules for sequence expressions are straightforward, with the only exception of the *seq-collect* expression. The type of this expression, often called *map* in other languages, is the type of the expression e, evaluated in an environment extended with the bound $\{id : \tau\}$.

$$\frac{\gamma, \varepsilon \vdash e : [\tau]}{\gamma, \varepsilon \vdash \mathbf{head} \ e : \tau} \quad (\text{seq-head}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau]}{\gamma, \varepsilon \vdash \mathbf{tail} \ e : [\tau]} \quad (\text{seq-tail}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : \tau, e' : [\tau]}{\gamma, \varepsilon \vdash e \ \mathbf{cons} \ e' : [\tau]} \quad (\text{seq-cons}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau]}{\gamma, \varepsilon \vdash \mathbf{isempty} \ e : Bool} \quad (\text{seq-isempty}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau], e' : [\tau]}{\gamma, \varepsilon \vdash e \textbf{ append } e' : [\tau]} \quad (\text{seq-append}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau], e' : [\tau]}{\gamma, \varepsilon \vdash e \textbf{ intersect } e' : [\tau]} \quad (\text{seq-intersect}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau], e' : [\tau]}{\gamma, \varepsilon \vdash e \textbf{ union } e' : [\tau]} \quad (\text{seq-union}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau], e' : [\tau]}{\gamma, \varepsilon \vdash e \textbf{ difference } e' : [\tau]} \quad (\text{seq-diff}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : \tau, e' : [\tau]}{\gamma, \varepsilon \vdash e \textbf{ isin } e' : Bool} \quad (\text{seq-isin}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [[\tau]]}{\gamma, \varepsilon \vdash \textbf{flatten } e : [\tau]} \quad (\text{seq-flat}\binom{exp}{stat})$$

$$\textbf{emptyseq of } \tau : [\tau] \quad (\text{seq-emptyseq}\binom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e' : [\tau] \quad \gamma, \varepsilon \cup \{id : \tau\} \vdash e : [\tau']}{\gamma, \varepsilon \vdash \textbf{collect } |id| \ e \textbf{ from } e' : \tau'} \quad (\text{seq-collect}\binom{exp}{stat})$$

Dynamic Semantics

We assume that the underlying virtual machine is equipped with the concept of sequence and we denote a sequence semantic value with a set of values enclosed by underlined square parenthesis.

$$\frac{Gamma, M \vdash e_i \Rightarrow v_i^{i \in 1..n}}{[e_1, \ldots, e_n] \Rightarrow \underline{[v_1, \ldots, v_n]}} \quad (\text{seq}\binom{exp}{dyn})$$

The results of the dynamic evaluation of sequence expressions is usually a semantic sequence value already introduced in **??**. We assume to have the following semantic operations on such values.

- $\underline{first}$(s): returns the first value v_1 of the sequence $[v_1, \ldots, v_n]$. For instance $\underline{first}([v_1, v_2]) = v_1$.

- $\underline{rest}$(s): returns a sequence created by removing the first element from the given sequence s, so that, for instance, $\underline{rest}([v_1, v_2, v_3]) = [v_2, v_3]$.
- $\underline{size}$(s): returns the number of elements of the sequence as a numeric value.
- $\underline{+}$(s, s'): concatenates two sequences.
- $\underline{cons}$(v, s): insert v in s as the new first element.
- $\underline{\cup}$(s, s'): performs the set union of two sequences.
- $\underline{\cap}$(s, s'): performs the set intersection of two sequences.
- $\underline{-}$(s, s'): performs the set difference of two sequences.

With these operations we can define the dynamic semantic of sequences. The following expressions are intuitive, with the possible exception of the *collect* expression. Such expression, often called *map* in other programming languages, takes in input an identifier id, an expression e and a sequence s. The elements of s are iterated and the identifier id is bound to each of such elements. The value of e is calculated in an environment extended with the new id for each of the sequence elements. Those values are packed in a new sequence in the order in which they are evaluated and returned as the result of the expression.

To deal with set operations, *intersection*, *union*, and *difference*, elements are compared by value, so that two elements are equal if and only if their value is equal.

The *flatten* expression takes in input a sequence s of sequences s_i and returns a sequence which elements are the elements of the elements of s in order of appearance. Note that the elements of the s_i sequences are left untouched, so that, if they are again a sequence, one or more other flattening can be necessary to obtain a flat sequence, that is, the *flatten* operation is not recursive.

$$\frac{\Gamma, M \vdash e \Rightarrow [v_1, \ldots, v_n]}{\Gamma, M \vdash \textbf{head}\ e \Rightarrow v_1} \quad (\text{seq-head}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow [v_1, v_2, \ldots, v_n]}{\Gamma, M \vdash \textbf{tail}\ e \Rightarrow [v_2, \ldots, v_n]} \quad (\text{seq-tail}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v_1, e' \Rightarrow [v_2, \ldots, v_n]}{\Gamma, M \vdash e\ \textbf{cons}\ e' \Rightarrow [v_1, v_2, \ldots, v_n]} \quad (\text{seq-cons}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash \textbf{isempty}\ e \Rightarrow \begin{cases} \underline{true} & if\ \underline{size}(v) = 0 \\ \underline{false} & else \end{cases}} \quad (\text{seq-isempty}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow [v_1, \ldots, v_n], e' \Rightarrow [v'_1, \ldots, v'_n]}{\Gamma, M \vdash e \textbf{ append } e' \Rightarrow [v_1, \ldots, v_n, v'_1, \ldots, v'_n]} \quad (\text{seq-append}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e \textbf{ intersect } e' \Rightarrow v \underline{\cap} v'} \quad (\text{seq-intersect}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e \textbf{ union } e' \Rightarrow v \underline{\cup} v'} \quad (\text{seq-union}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow v'}{\Gamma, M \vdash e \textbf{ difference } e' \Rightarrow v \underline{-} v'} \quad (\text{seq-diff}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow v, e' \Rightarrow [v_1, \ldots, v_n]}{\Gamma, M \vdash e \textbf{ isin } e' \Rightarrow \begin{cases} \underline{true} & v \in \{v_1, \ldots, v_n\} \\ \underline{false} & else \end{cases}} \quad (\text{seq-isin}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow [[v^1_1, \ldots, v^1_{n_1}], \ldots, [v^m_1, \ldots, v^m_{n_k}]]}{\Gamma, M \vdash \textbf{flatten } e \Rightarrow [v^1_1, \ldots, v^1_{n_1}, \ldots, v^m_1, \ldots, v^m_{n_k}]} \quad (\text{seq-flat}\tbinom{exp}{dyn})$$

$$\textbf{emptyseq of } \tau \Rightarrow \underline{[]} \quad (\text{seq-emptyseq}\tbinom{exp}{dyn})$$

$$\frac{\begin{array}{r}\Gamma, M \vdash e' \Rightarrow [v_1, \ldots, v_n] \\ \Gamma \cup \{id = v_i\}, M \vdash e \Rightarrow v_i \ \forall\ i \in 1 \ldots n\end{array}}{\Gamma, M \vdash \textbf{collect } |id| \ e \textbf{ from } e' \Rightarrow [v_i] \ \forall\ i \in 1 \ldots n} \quad (\text{seq-collect}\tbinom{exp}{dyn})$$

5.5.18 Polymorphism

The `Poly` bundle, introduced in this section, extends Manuzio with the concept of parametric polymorphism. With parametric polymorphism a portion of code, usually a function, can be applied to different data types that can be non-compatible. In a statically, strongly typed language like Manuzio the type checking imposes each value to have a type. Since functions are also values, without parametric polymorphism it would be impossible to write a trivial function like the identity function without writing one function for each type of the language to which we plan to apply it. For instance, the identity functions for integers and reals follows.

```
1         let intIdentity = fun(x:Int):Int is x
2         let realIdentity = fun(x:Real):Real is x
```

By introducing the `Poly` bundle we will be able to write a generic identity function as:

```
1         let identity = polyfun [T] is fun(x:T):T is x
```

where T is a generic type. To use a parametric function we must first instantiate it with one specific type. The results of such operation can be stored in a constant for later use or used on the fly.

```
1         let intIdentity = @identity[Int]
2         @intIdentity(2) #=> 2:Int
3         @(@identity[Real])(1.5) #=> 1.0:Real
```

To be able to perform the instantiation of specific functions from generic ones we need to substitute the formal type parameters with the actual ones by using the FTV rules that each type of the language defines.

Types Environment Extension

To represent the concept of generic type a new type, called *All*, is introduced to represent the type of a polymorphic function. Such type is composed by a function of type τ_f and one or more type variables X_i and represents a type schema that can be instantiated by substituting the type variables with actual types in the function body.

$$\gamma' = \gamma \cup \{All[X_1, \ldots, X_n]\tau_f\} \tag{5.37}$$

Syntax

A polymorphic function is denoted by an expressions starting with the *polyfun* keyword. Such definitions takes in input a set of type variables X_i and an expression

e, that must type check as a function later, in which such type variables will be substituted. The value of such expression is a polymorphic function type.

$$polyfun[X_1, \ldots, X_n] \text{ is } e \qquad \text{(poly-fun)}$$

The only operation an user can carry on a polymorphic function is to instantiate it with an opportune set of actual type parameters. This is done through the *apply* operator denoted by the @ symbol.

$$@e[\tau_1, \ldots, \tau_n] \qquad \text{(poly-app)}$$

Static Semantics

Free Type Variables and Type Variables Substitution

When we compute the free type variables of a polymorphic type we compute the f.t.v. of its body function type τ_f and we exclude the types given as a parameter.

$$FTV(All[X_1, \ldots, X_n]\tau_f) = FTV(\tau_f) - \{X_1, \ldots, X_n\} \qquad \text{(poly}\tbinom{ftv}{stat}\text{)}$$

Substitutions in the function body τ_f occurs by applying the substitutions rules introduced by each bundle.

$$(All[X_i, \ldots, X_n]\tau_f)[X_i \leftarrow \tau_i^{i\in 1..n}] = \tau_f[X_i \leftarrow \tau_i^{i\in 1..n}] \qquad \text{(poly}\tbinom{subs}{stat}\text{)}$$

The substitution is carried out in two steps: the first will take care of avoiding the substitution of bound type variables, while the second will prevent the capture of free type variables.

1. In the first step we must check if any type variable to be substituted has the same name as a formal type parameter. In this case the substitution must not be carried out.

$$\begin{aligned}&(All[Y_j^{j\in 1\ldots m}]\tau_f)[X_i \leftarrow \tau_i^{i\in 1\ldots n}] = \\ &(All[Y_j^{j\in 1\ldots m}]\tau_f)[X_i \leftarrow \tau_i^{\{i\in 1\ldots n\}-\{k\ldots l\}}] = \\ &if\ \{Y_1, \ldots, Y_m\} \cap \{X_1, \ldots, X_n\} = \{X_k, \ldots, X_l\}\end{aligned} \qquad (5.38)$$

2. The second step avoids the free type variables capture and we can distinguish two cases. In the first one the names of the parameters $Y_1, \ldots Y_n$ and the

names of the free variables $\tau_1, \ldots, \tau_n$ are different and the substitution can be defined as follows.

$$(All[Y_j^{j\in 1\ldots m}]\tau_f)[X_i \leftarrow \tau_i^{i\in 1\ldots n}] = \tau_f[X_i \leftarrow \tau_i^{i\in 1..n}]$$
$$if\ \{Y_1, \ldots, Y_m\} \cap \{\cup_{i\in 1\ldots n} FTV(\tau_i) = \{\} \tag{5.39}$$

In the second case instead at least one actual type variable with the same name as a free type variable exists. Such variable must be renamed to avoid its capture, and this is done by a function called *newvar* that returns a new variable name that can be used to perform such substitution.

$$(All[Y_j^{j\in 1\ldots m}]\tau_f)[X_i \leftarrow \tau_i^{i\in 1\ldots n}] =$$
$$(All[Y_j^{j\in 1\ldots m}]\tau_f)[Y_k \leftarrow Z_1, \ldots, Y_l \leftarrow Z_r])[X_i \leftarrow \tau_i^{i\in 1\ldots n}]$$
$$if\ \{Y_1, \ldots, Y_m\} \cap \{\cup_{i\in 1\ldots n} FTV(\tau_i) = \{Y_k, \ldots Y_l\} \tag{5.40}$$

where k and l are between 1 and m, the cardinality of $\{Y_k, \ldots Y_l\}$ is r and

$$\forall i \in 1 \ldots r : Z_i = newvar() \tag{5.41}$$

Well Formedness

A polymorphic type is well formed if it's body is well formed, under the assumption that each type parameter name is associated with a generic type. We will introduce a new support type, called *Generic* just to denote such concept in the following formula.

$$\frac{\gamma \cup \{X_i \leftarrow Generic\}^{i\in 1\ldots n}, \varepsilon \vdash \tau_f : Fun(\tau_1, \ldots, \tau_n) : \tau_{n+1}}{\gamma, \varepsilon \vdash All[X_i^{i\in 1\ldots n}]\tau_f} \quad (\text{poly}\tbinom{well}{stat})$$

Subtyping Relations

A polymorphic type τ is subtype of another type τ' if and only if τ' is a polymorphic type and the type of body of τ is subtype of the type of the body of τ' under the assumption that the type variables of τ' have been substituted with the type variables of τ.

$$\frac{\gamma \cup \{X_i \leftarrow Generic\}^{i\in 1\ldots n}, \varepsilon \vdash \tau_f <: \tau'_f[Y_i \leftarrow X_i]^{i\in 1\ldots n}}{\gamma, \varepsilon \vdash All[X_i^{i\in 1\ldots n}]\tau_f <: All[Y_i^{i\in 1\ldots n}]\tau'_f} \quad (\text{poly}\tbinom{sub}{stat})$$

Free Identifiers and Visibility Rules

In a polymorphic abstraction the free identifiers are the free identifiers of its body.

$$FV(polyfun[X_1, \cdots x_n]ise) = FV(e) \qquad (\text{poly}\binom{fv}{stat})$$

When a polymorphic function is instanced, the free identifiers are the ones of the polymorphic abstraction to be instantiated.

$$FV(@e[X_1, \ldots, X_n]) = FV(e) \qquad (\text{poly-app}\binom{fv}{stat})$$

Type Checking Rules

Given all the previous rules the type checking rules for the instantiation of a polymorphic function are rather simple. If the expression e is a polymorphic function then the instantiation of e with a set of types τ_i is a the type of the body of e where the type variables have been substituted.

$$\frac{\gamma, \varepsilon \vdash e : All[X_i^{i \in 1 \ldots n}]\tau}{\gamma, \varepsilon \vdash @e[\tau_1, \ldots, \tau_n] : \tau[X_i \leftarrow \tau_i]^{i \in 1 \ldots n}} \qquad (\text{poly-app}\binom{exp}{stat})$$

Dynamic Semantics

The evaluation of a polymorphic function returns the value of that function's body.

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash polyfun[X_1 \ldots X_n]ise \Rightarrow v} \qquad (\text{poly}\binom{exp}{dyn})$$

The evaluation of a polymorphic function instantiation is the value of a function, defined as follows.

$$\frac{\Gamma, M \vdash e \Rightarrow v}{\Gamma, M \vdash @e[\tau_1, \ldots, \tau_n] \Rightarrow v} \qquad (\text{poly-app}\binom{exp}{dyn})$$

5.5.19 Objects

The `Objects` bundle extends the Manuzio programming language with a simple form of objects. In Manuzio an object is a software entity with an identity, a state, and a behavior. The state is a set of values called its instance variables, while the behavior is a set of local functions, called methods. A method is a special kind of function, syntactically denoted by the keyword *meth*, that uses instance variables and eventually parameters to produce a result. The identity of an object is an immutable property that makes the object different from any other object. Such identity is set when the object is created and never changes, even if the values of the object's instance variables change. The result is that two objects with the same values of instance variables and methods are still different. An object can receive a set of *messages*, to which the object responds with the value computed by one of its methods. The type of an object is also called its *interface*, and lists all the messages that an object can respond to. The semantic value of an object is a tuple of instance variables and methods. In Manuzio objects lack of encapsulation, mainly because they are designed to be the foundation of textual objects, where such feature is not essential.

Types Environment Extension

An object type is specified by the keyword *OBJECT* followed by the tuple of its instance variables and methods.

$$\gamma' = \gamma \cup \{\textbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \textbf{end}\} \tag{5.42}$$

Syntax

An object value is denoted by the *object* keyword followed by a list of instance variables and methods terminated by the *end* keyword.

$$\textbf{object}\ id_i = e_i^{i \in 1..n}, id_j = meth(x_k : \tau_k^{k \in 1..l}) : \tau_j\ \textbf{is}\ e_j^{j \in 1..m}\ \textbf{end} \tag{object}$$

The main object operator is the *dot* operator, used to access its properties (instance variables and methods).

$$e.id \tag{object-dot}$$

The *objectextend* operator, used to extend an object with new instance variables and methods, is similar in syntax and logic to *extend* of records.

$$e\ \textbf{objectextend}\ \{id_1 : \tau_1 = e_1, \ldots, id_n : \tau_n = e_n\} \tag{object-ext}$$

Static Semantics

Free Type Variables and Type Variables Substitution

The free type variables of an *OBJECT* type are defined as the union of the free type variables of the types of its fields.

$$FTV(\textbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \textbf{end}) = FTV(\tau_1) \cup \ldots \cup FTV(\tau_n) \quad (\text{object}\binom{ftv}{stat})$$

In the same way, the substitutions that occur in an object type are treated as substitutions in their fields.

$$\begin{aligned}(\textbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \textbf{end})[X_i \leftarrow \tau_i]^{i \in 1 \ldots n} = \\ \textbf{OBJECT}\ id_1 : \tau_1[X_i \leftarrow \tau_i], \ldots, id_n : \tau_n[X_i \leftarrow \tau_i]\ \textbf{end}^{i \in 1 \ldots n}\end{aligned} \qquad (\text{object}\binom{sub}{stat})$$

Well Formedness

An object type is well formed when:

- the identifiers of its fields are valid;
- each identifier is unique;
- the types of its fields is well formed.

$$\frac{\forall i \in (1..n)\ id_i \notin (\{id_1, \ldots, id_n\} - \{id_i\}), id_i \in \iota, \tau_i}{\gamma, \varepsilon \vdash \textbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \textbf{end}} \qquad (\text{object}\binom{well}{stat})$$

Subtyping Relations

Two object types O and O' are in a subtype relation if O has at least the same number of fields as O' and if, for each field of O' with type τ' exists in O a field with the same name and type τ that is subtype of τ'.

$$\frac{\gamma, \varepsilon \vdash \{id_1 : \tau_1, \ldots, id_n : \tau_n\} <: \{id'_1 : \tau'_1, \ldots, id'_m : \tau'_m\}}{\gamma, \varepsilon \vdash \textbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \textbf{end} <: \textbf{OBJECT}\ id'_1 : \tau'_1, \ldots, id'_m : \tau'_m\ \textbf{end}} \qquad (\text{object}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

The free identifiers of objects operators are computed as the free identifiers of their relative records. The object-fv equation is needed to avoid the capture of the self reference parameter identifier in the body of object's methods. Unlike recursive functions, where the self reference identifier was declared by users, in object its name is fixed to *self*.

$$FV(\mathbf{object}(e_r)) = FV(e_r) - \{self\} \qquad \text{(object-fv)}$$

$$FV(e.id) = FV(e) \qquad \text{(object-dot}\tbinom{fv}{stat}\text{)}$$

$$FV(e \ \mathbf{objectextend}\ \{id_1 : \tau_1 = e_1, \ldots, id_n : \tau_n = e_n\}) = \bigcup_{i \in 1 \ldots n} FV(e_i) \cup FV(e) \qquad \text{(object-ext}\tbinom{fv}{stat}\text{)}$$

Type Checking Rules

When computing the type of an object, the type of the *self* identifier is firstly computed by constructing an object type T where we assume correct the return type of methods as it is declared. With this assumption the methods bodies can be typechecked correctly. If all the methods pass the check then the precomputed object type T is considered correct and is returned as the type of the object.

$$\frac{\gamma, \varepsilon \vdash e_i : \tau_i^{i \in 1 \ldots n}, \gamma, (self : \tau_o) \vdash e_j : \tau_j^{j \in n+1 \ldots m}}{\begin{array}{c}\gamma, \varepsilon \vdash \mathbf{object}\ id_i = e_i^{i \in 1 \ldots n}, id_j = meth(x_k : \tau_k^{k=1..l}) : \tau_j\ \mathbf{is}\ e_j^{j \in n+1 \ldots m}\ \mathbf{end}\ : \\ \mathbf{OBJECT}\ id_h : \tau_h^{h \in 1..m} \mathbf{end}\end{array}} \qquad (5.43)$$

When accessing a component with the dot notation the type of the expression is the type of the selected component, if such component exists.

$$\frac{\gamma, \varepsilon \vdash e : \mathbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \mathbf{end}, id = id_i^{i \in 1..n}}{\gamma, \varepsilon \vdash e.id : \tau_i} \qquad \text{(object-dot}\tbinom{exp}{stat}\text{)}$$

The extension operator yields a new object with additional fields, redefinition of existing fields is not allowed.

$$\frac{\begin{array}{c}\gamma, \varepsilon \vdash e : \mathbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n\ \mathbf{end} \\ \gamma, \varepsilon \vdash id'_1 : tau'_1, \ldots, id'_m : \tau'_m,\ id_i \neq id'_j\ \forall\ (i \in 1 \ldots n, j \in 1 \ldots m)\end{array}}{\begin{array}{c}\gamma, \varepsilon \vdash e\ \mathbf{objectextend}\ \{id'_1 : \tau'_1, \ldots, id'_m : \tau'_m\} \\ : \mathbf{OBJECT}\ id_1 : \tau_1, \ldots, id_n : \tau_n, id'_1 : \tau'_1, \ldots, id'_m : \tau'_m\ \mathbf{end}\end{array}}$$

$$(\text{object-ext}\binom{exp}{stat})$$

Dynamic Semantics

The value of an object is defined as:

$$\lambda O.\left[\begin{array}{c} x_i^{i\in 1\ldots n} = v_i, \\ x_j^{j\in n+1\ldots m=\langle(\Gamma'\cup(self=O)),e_j\rangle} \end{array}\right] \tag{5.44}$$

where each label x_k is distinct. Object values are written in a way similar to the lambda notation so that the object itself can be denoted when associated to the self-reference identifier. For each method x_j the closure Γ' is constructed as:

$$\Gamma' = \{(z_i = v_i)^{i\in 1..j} : z_i \in FV(e_j) - \{self\}\} \tag{5.45}$$

The evaluation of an object constant can thus be written as:

$$\frac{\Gamma, M \vdash e_i \Rightarrow v_i}{\textbf{object}\ id_i = e_i^{i\in 1\ldots n}, id_j = meth : \tau_j\ \textbf{is}\ e_j^{j\in 1\ldots m}\ \textbf{end} \Rightarrow \lambda O.\left[\begin{array}{c} x_i^{i\in 1\ldots n} = v_i, \\ x_j^{j\in n+1\ldots m=\langle(\Gamma'\cup(self=O)),e_j\rangle} \end{array}\right]} \tag{5.46}$$

The dot operator returns the value of the corresponding label. Such value can be a closure if the label corresponds to a method.

$$\frac{\Gamma, M \vdash e \Rightarrow \lambda O.\left[\begin{array}{c} x_i^{i\in 1\ldots n} = v_i, \\ x_j^{j\in n+1\ldots m=v_j=\langle(\Gamma'\cup(self=O)),e_j\rangle} \end{array}\right] \exists k^{k\in 1..m} : id = x_k}{\Gamma, M \vdash e.id \Rightarrow v_k}$$

$$(\text{object-dot}\binom{exp}{dyn})$$

Object's extensions are similar to the record's extend operation.

$$\frac{\Gamma, M \vdash \textbf{object}\ id_1^o = v_1^o, \ldots, id_n^o = v_n^o\ \textbf{end}}{\begin{array}{c}\Gamma, M \vdash e_1 \Rightarrow v_1, \ldots, e_m \Rightarrow v_m \\ \Gamma, M \vdash e\ \textbf{objectextend}\ \{id_1 = e_1, \ldots, id_m = e_m\} \Rightarrow \\ \textbf{object}\ id_1^o = v_1^o, \ldots, id_n^o = v_n^o, id_1 = v_1, \ldots, id_m = v_m\ \textbf{end}\end{array}}\ (\text{object-ext}\binom{exp}{dyn})$$

5.5.20 Textual Objects

The `Textual Object` bundle introduces the concept of textual objects as a special kind of objects that are used to represent a portion of text taken from a special repository called *textual database*. In this section the term "textual object" will be used to indicate both single and repeated textual objects. When dealing with textual objects, in fact, the goal of the language is to treat both single textual objects and their repetitions in a *transparent* way, through the use of overloaded operators. We will call these operators *textual objects operators*.

The `Textual Object` bundle exploit the functionalities of the Manuzio persistence layer. An user can, through the use database (*usedb*) command, connect to a textual database. When such command is launched with a parameter d the persistence layer checks for the existence of the database d first, then, if the database exists, performs the connection. By doing so all the textual object types specified by the database model are loaded into the types environment and a special constant *collection* is created in the values environment. This constant is always of type `Collection` and its underlying text is the entire text. Such constant serves as an entry point for the entire corpus, since by applying the hierarchy navigation operators on it it will be possible to reference the whole set of the database's textual objects.

Textual objects types are, in general, not declare by users at runtime. Such types are loaded into the environments during the database's connection process. In this way the textual objects types present in the language at a given time form always a well-formed Manuzio model. Also, textual object type instances cannot be created or deleted, they already exists as persistent values stored in the textual database. A class of Manuzio programs, containing the special textual schema declaration block, allow users to define textual object types and to use a set of textual object creation expressions to write programs that parse an input text into a textual repository. The syntax and semantics of these operators is not included in this section.

Textual objects can be annotated with the *annotate* operator. Annotations are persistent and can be made both on single or repeated textual objects. The type of an annotation is specified in the textual database's model, so that they can be treated as values of the Manuzio language directly with all the associated benefits, such as typechecking consistency.

Types Environment Extension

A textual object type is a tuple type structured as follows:

$$TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\} \tag{5.47}$$

in particular, the type of a single textual object that has no elements can be denoted by:

$$TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : NULL \end{array} \right\} \tag{5.48}$$

while the type of a repeated textual object that has no components and indexes can be denoted by:

$$TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : NULL \\ c : NULL \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\} \tag{5.49}$$

Persistence Model

To define textual objects semantics it is first needed to define the concept of *textual repository*. A textual repository is an abstract entity that behave in a manner similar to a memory, but is meant to store persistently textual object values.

Definition 24 *A textual repository is a pair (f, R), were f is a text called the fulltext, and R is a sequence of locations such that:*

1. *The pair $(f, \emptyset)$ is a valid textual repository.*
2. *If (f, R) is a valid textual repository, l is a location index and t is a textual object value then $(f, R) \cup \{l = t\}$ is a valid textual repository.*

If $l = t$ is an element of R we can write $(l = t) \in R$ or $R(l) = t$. Each l is unique in R, and the value of l is said to be the address of t and we can refer to t as t_l. The notation $R(l) \leftarrow t$ indicates that the object t gets stored in the location l.

Each location of R is identified by an address and can contain exactly one textual object semantic value. The textual object semantic values set is partitioned in two subsets, one composed of single textual object semantic values and the other one composed of repeated textual object semantic values.

The textual repository has an associated procedure, $h(t)$, that returns the address of an empty cell where the textual object t must be stored. The $h(t)$ procedure is an hash function that, given the unique characteristics of an object t, returns the textual repository location l for t. With unique characteristics we intend the pair $(tc, indexes)$ for single textual objects and the sequence $[e_1, \ldots, e_n]$ for repeated textual objects[1]. We assume, without loss of generality, that the function h is an ideal hash function, so that it always returns a different location for each possible textual object in the database. The value of $h(t)$ is also called the address of t. From the language point of view, a textual object value t is represented by the address returned by $h(t)$, that is used to access the textual repository. We will refer to the textual object with address l as t_l, or we can write $R(l) = t_l$.

We assume that single textual objects are already present in the database, along with repeated textual objects that posses annotations. When a new repeated textual object's value is returned, for instance as the result of a query, that value needs to be stored in the repository. The $h(t)$ procedures is invoked and the value of the repeated textual object is stored in the returned address's cell.[2].

The textual objects values contained in the textual repository cells are tuples, composed of the following fields:

- **kind**: the kind field serves as a discriminant between single and repeated textual objects;
- **typecode**: an encoded representation of the object's type;
- **indexes**: a sequence of pairs (start, offset) that represents chunks of the full text;
- **components**: a record of n components which fields are all textual objects addresses, in the form $\{c_1 = l_1, \ldots, c_n = l_n\}$;

[1] In fact, two single textual object values will be considered equals if and only if their textual indexes and their type code are both equals, while two repeated textual objects are equals if they have exactly the same set of elements.

[2] Storing repetitions in the database is needed to store their annotations. For efficiency reasons, however, only the annotated and referenced repetitions are stored. When a textual database connection is closed, the textual repository is purged of all the instances of repeated textual objects that are not annotated. While this consideration seems to be inappropriate in an abstract context such this, we decided, in this borderline case, to avoid too much abstractions in favor of an easier implementation.

- **attributes**: a record of p attributes in the form $\{a_1 = v_1, \ldots, a_m = v_m, a_{m+1} = meth(x_i : \tau_i^{i\in 1..o}) : \tau_{m+1}, \ldots, a_p = meth(x_i : \tau_i^{i\in 1..o}) : \tau_p\}$;

- **elements**: a sequence of textual objects in the form $[e_1, \ldots, e_k]$;

When the *kind* field assumes the 0 value the tuple is considered to represent a single textual object value, while when it assumes the 1 value it is considered to represent a repeated textual object value. In the former case the *elements* field is set to a null value, since single textual objects do not have elements. In the latter case, instead, both the *indexes* and *components* fields are set to null. Our semantics for textual objects is close to the one of objects presented in Section 5.5.19. For this reason the semantics of methods will not be repeated here. It is sufficient to say that for each method x_k the closure Γ' enclose both attributes and components of a textual object, and is constructed as follows:

$$\Gamma' = \{(c_i = l_i)^{i\in 1..n}) \cup (a_j = v_j)^{j\in 1..k} : z_i \in FV(e_k) - \{self\}\} \tag{5.50}$$

A textual object's semantic value will be denoted with:

$$\lambda TO. \begin{bmatrix} \text{kind} \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_n, o_n)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{bmatrix} \tag{5.51}$$

Textual object values are written in a way similar to the lambda notation so that the object itself can be denoted when associated to the self-reference identifier of its associated methods. An abbreviated version of the above formula can be used to denote fields in a more concise way:

$$\lambda TO. \begin{bmatrix} \text{kind} \\ \text{typecode} \\ \text{indexes} \\ c \\ a \\ e \end{bmatrix} \tag{5.52}$$

In particular, a single textual object is denoted by:

$$\lambda TO. \begin{bmatrix} 0 \\ \text{typecode} \\ \text{indexes} \\ c \\ a \\ nil \end{bmatrix} \tag{5.53}$$

while a repetition is denoted by:

$$\lambda TO. \begin{bmatrix} 1 \\ \text{typecode} \\ nil \\ nil \\ a \\ e \end{bmatrix} \tag{5.54}$$

When an operator returns a repeated textual object, for instance by aggregating two or more single textual objects, the interpreter must perform a reverse look-up in the textual repository to know if that object is already present at the relative location. For instance, if the returned repeated textual object's value is in the form:

$$t' = \lambda TO. \begin{bmatrix} 1 \\ \text{typecode} \\ nil \\ nil \\ a \\ [e_1, \ldots, e_n] \end{bmatrix} \tag{5.55}$$

the object address l is computed by calling $h(t')$. This address is used to lookup the textual repository. If $R(l) = t$ then the repetition is already present in the textual repository, possibly with some annotations a, and the value t is returned by the operator instead of the unannotated t'. Else, if the cell at the address l is empty, the value is not yet present and the location gets initialized with the value of t' that is also returned as result.

The fulltext f can be considered, without loss of generality, to be simply a monolithic string that contains all the characters of the corpus in lexicographic order. We can apply the slice operator to f so that $f[i..j]$ means the substring of f that starts from the i-nth character and ends on the j-nth character (included).

Syntax

Textual objects can not be created by the language users, to the Manuzio language, at the current stage of development, lacks of expressions to perform such operations.

The `Textual Object` bundle introduces a collection of high level operators to deal with textual objects components and attributes. Most of those operators are overloaded so that, from the interpreter point of view, both attributes and components are handled in the same way. We felt that this transparency is helpful to the user when dealing with literary analysis problems.

The most important operators of the bundle are the *get* and *getall* operators. They are used to retrieve textual objects from the database by exploiting the component relation.

get *id* **of** *e* (to-get[c—a][s—r])

The result type the *get* operator is the type of the attribute or component specified in the textual object type interface. In the case of components the result type can be either a textual object or a repeated textual object.

getall T **of** *e* (to-get[c—a][s—r])

The *getall* operator, instead, works on textual objects and is used to retrieve all the components of a certain type T (direct or indirect) of a textual object. The *getall* operator, instead, always returns a repeated textual object.

Both the *get* and the *getall* operators are overloaded. Each is present in four different fashions to that is can work on components/attributes and on singe/repeated textual objects. The formula name will contain a c or an a to discriminate the former and an s or a r for the former.

Another family of operators allow to access other, calculated, textual object features, most importantly its underlying text, with the *text of* operator. As with the *get* operator family, such operator is overloaded to be able to work both on repetitions and on single textual objects. In the language other operators, like the *size of* one to obtain the length of a textual object underlying text, are in fact shortcuts to work directly with the object underlying text in a simpler way and thus their semantics will be skipped.

text of *e* (to-textof[s—r])

Finally, the *annotate .. set .. to* operator takes in input a textual object o, a label l and a value v and set the annotation l of o to v. In Manuzio all the annotations of a textual object are always associated to a value. This value, however, can be unknown, assuming the value *nil* of type `NULL`. Both single and repeated textual objects can be annotated, and the annotation must be part of the textual object type declaration to be valid.

annotate *e* **set** *id* **to** e' (to-ann)

The set of textual object operators, in this release of the language, is minimal. It will be a future decision if it is better to expand this set directly of to relay on libraries to obtain more complex functionalities.

Static Semantics

Free Type Variables and Type Variables Substitution

The set of free type variables of a textual object type is the union of the free type variables of its fields. For brevity, in the following formula, we omit to add the fields that has a basic type, and thus an empty set of free type variables.

$$FTV(TO\left\{\begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array}\right\}) = FTV(\{c_i : \tau_i\}^{i \in 1..n}) \cup FTV(\{a_j : \tau_{a_j}\}^{j \in 1..m}) \cup FTV([\tau_e]) \quad (\text{to}\binom{ftv}{stat})$$

Note that the formula is valid both for single and repeated textual objects, since $FTV(NULL) = \emptyset$.

Well Formedness

A textual object type is well formed if and only if both its components and attributes tuple types are well formed. Moreover, labels must be unique among those two types. Moreover, to enforce the well-formedness of repeated textual objects, the sequence of elements must have also a well-formed type.

$$\frac{\begin{array}{c}\gamma, \varepsilon \vdash \{c_i : \tau_i\}^{i \in 1..n}, \{a_j : \tau_{a_j}\}^{j \in 1..m}, [\tau_e] \\ \forall i \in (c_1 \ldots c_n \cup a_1 \ldots a_n) : id_i \text{ is unique}\end{array}}{TO\left\{\begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array}\right\}} \quad (\text{to}\binom{well}{stat})$$

Subtyping Relations

A textual object type T is a subtype of another textual object type T' if their components, attributes, and elements types are in a subtype relation.

$$\frac{\begin{array}{c}\gamma, \varepsilon \vdash \{c_i : \tau_i\}^{i \in 1..n} <: \{c'_i : \tau'_i\}^{i \in 1..n'}, \\ \{a_j : \tau_{a_j}\}^{j \in 1..m} <: \{a'_j : \tau'_{a'_j}\}^{j \in 1..m'}, \\ [\tau_e] <: [\tau'_e]\end{array}}{TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\} <: TO \left\{ \begin{array}{l} kind' : Int, \\ tc' : Int, \\ i' : [\{s' : Int, o' : Int\}] \\ c' : \{c'_i : \tau'_i\}^{i \in 1..n'} \\ a' : \{a'_j : \tau'_{a'_j}\}^{j \in 1..m'} \\ e' : [\tau'_e] \end{array} \right\}} \quad (\text{to}\binom{sub}{stat})$$

Free Identifiers and Visibility Rules

The rules to compute the free identifiers of textual objects related expressions are straightforward.

$$FV(\textbf{get } id \textbf{ of } e) = FV(e) \qquad (\text{to-get[c—a][s—r]}\binom{fv}{stat})$$

$$FV(\textbf{getall } T \textbf{ of } e) = FV(e) \qquad (\text{to-getall[c—a][s—r]}\binom{fv}{stat})$$

$$FV(\textbf{text of } e) = FV(e) \qquad (\text{to-text[s—r]}\binom{fv}{stat})$$

$$FV(\textbf{annotate } e \textbf{ set } id \textbf{ to } e') = FV(e) \cup FV(e') \qquad (\text{to-ann}\binom{fv}{stat})$$

Type Checking Rules

Type checking rules apply both to single and repeated textual objects. The *get .. of* operator returns the type of a component or annotation when applied to a single textual object.

$$\frac{\gamma, \varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\}, \exists k : id = c_k^{k \in (1..n)} \text{ or } id = a_k^{k \in 1..m)} : T}{\gamma, \varepsilon \vdash \textbf{get } id \textbf{ of } e : T}$$

$$(\text{to-get[c—a][s]}\binom{exp}{stat})$$

When applied to repetitions, instead, it returns the type of one of its annotations if the given identifier is part of the annotations tuple.

$$\frac{\gamma, \varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\}, \exists k : id = a_k^{k \in 1..m)} : T}{\gamma, \varepsilon \vdash \textbf{get } id \textbf{ of } e : T} \; (\text{to-get[a][r]}\binom{exp}{stat})$$

If the identifier is not an annotation, instead, the returned type is a repeated textual object which type is taken from the argument element's components, provided that it exists.

$$\frac{\gamma, \varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : NULL \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau] \end{array} \right\}, \tau = TO' \left\{ \begin{array}{l} kind' : Int, \\ tc' : Int, \\ i' : [\{s' : Int, o' : Int\}] \\ c' : \{c'_i : \tau'_i\}^{i \in 1..n'} \\ a' : \{a'_j : \tau'_{a_j}\}^{j \in 1..m'} \\ e' : NULL \end{array} \right\} \exists k : id = {c'_k}^{k \in 1..n}}{\gamma, \varepsilon \vdash \textbf{get } id \textbf{ of } e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : NULL \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_k] \end{array} \right\}} \; (\text{to-get[c][r]}\binom{exp}{stat})$$

The getall operator, instead, always returns a, possibly empty, repeated textual object of the specified type.

$$\frac{\gamma, \varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\}}{\gamma, \varepsilon \vdash \textbf{getall } T \textbf{ of } e : REP[T]} \quad \text{(to-getall[c][s—r]}\tbinom{exp}{stat}\text{)}$$

The *text .. of* operator, when applied to a textual object, returns a string.

$$\frac{\gamma, \varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\}}{\gamma, \varepsilon \vdash \textbf{text of } e : String} \quad \text{(to-text[s—r]}\tbinom{exp}{stat}\text{)}$$

Finally, the *annotate .. set .. to* operator requires that the specified label is in effect a label of the textual object associated with an annotation. Moreover, the type of the specified expression must be compatible with the declare annotation's type .

$$\frac{\gamma\varepsilon \vdash e : TO \left\{ \begin{array}{l} kind : Int, \\ tc : Int, \\ i : [\{s : Int, o : Int\}] \\ c : \{c_i : \tau_i\}^{i \in 1..n} \\ a : \{a_j : \tau_{a_j}\}^{j \in 1..m} \\ e : [\tau_e] \end{array} \right\}, e' : \tau, \exists j^{j \in 1..m} ! id = a_j, \tau <: \tau_j}{\gamma, \varepsilon \vdash \textbf{annotate } e \textbf{ set } id \textbf{ to } e' : Command}$$

$$\text{(to-ann[s—r]}\tbinom{exp}{stat}\text{)}$$

Dynamic Semantics

With the previous definition of textual repository we can give the dynamic semantics of textual objects. Differently from the previous sections, where operators families (like the *get* operator) were grouped, in this section, for the sake of readability, each different case of such operators will be described in its own formula. In the formula names the following convention is used: after the name of the operator the letter "c" indicates a component-related operator, while the letter "a" is used for operators

that deals with attributes. The second letter of the code is either an "s" or an "r": in the first case the operator applies to single textual objects, in the second to repeated textual objects. The equations to-texts$\binom{exp}{dyn}$ and to-textr$\binom{exp}{dyn}$ make use of the *strcat* semantic operator described in Section 5.5.14 to perform a concatenation of the object's underlying text.

The to-getallr$\binom{exp}{dyn}$ and to-getalls$\binom{exp}{dyn}$ formulas make use of a new recursive operator that, given a textual object and a type, returns all the object's direct and indirect components of that type. The formal definition of that operator can be found in 5.56.

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, \quad R(l) = \lambda TO. \begin{bmatrix} 0 \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_n, o_n)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ nil \end{bmatrix}}{\Gamma, M, R \vdash \textbf{get } c_i \textbf{ of } e \Rightarrow l_i} \quad (\text{to-getcs}\tbinom{exp}{dyn})$$

$$\frac{\begin{array}{c}\Gamma, M, R \vdash e \Rightarrow l, \\ R(l) = \lambda TO. \begin{bmatrix} 1 \\ \text{typecode} \\ nil \\ nil \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{bmatrix}, \\ t = \lambda TO'. \begin{bmatrix} 1 \\ \text{typecode'} \\ nil \\ nil \\ \{\} \\ [\textbf{get } c_i \textbf{ of } e_i] \end{bmatrix}, \\ h(t) = l' \end{array}}{\Gamma, M, R \vdash \textbf{get } c_i \textbf{ of } e \Rightarrow \begin{cases} t \diamond R(l') \leftarrow t, & \text{if } R(l') = nil \\ R(l') & \text{else} \end{cases}} \quad (\text{to-getcr}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} 0 \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_n, o_n)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ nil \end{array}\right]}{\Gamma, M, R \vdash \textbf{get } a_i \textbf{ of } e \Rightarrow v_i} \quad (\text{to-getas}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} 1 \\ \text{typecode} \\ nil \\ nil \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{array}\right]}{\Gamma, M, R \vdash \textbf{get } a_i \textbf{ of } e \Rightarrow v_i} \quad (\text{to-getar}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} 0 \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_n, o_n)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ nil \end{array}\right]}{\Gamma, M, R \vdash \textbf{text of } e \Rightarrow \underline{strcat}(fulltext[s_0, o_0], \ldots, fulltext[s_n, o_n])} \quad (\text{to-texts}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} 1 \\ \text{typecode} \\ nil \\ nil \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{array}\right]}{\Gamma, M, R \vdash \textbf{text of } e \Rightarrow \underline{strcat}(\textbf{text of } e_i)^{i \in 1..n}} \quad (\text{to-textr}\tbinom{exp}{dyn})$$

$$\rho(t, \tau) = \bigcup_{i=1}^{n} \begin{cases} v_i, & \text{if } v_i : \tau \\ \emptyset, & \text{if } \tau_i \in \text{ Unit} \\ \rho(v_i, \tau) & \text{else} \end{cases} \tag{5.56}$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = t_l}{\Gamma, M, R \vdash \textbf{getall } \tau \textbf{ of } e \Rightarrow h(\{1, tc, nil, \alpha(t), \{\}, [\rho(t_l, \tau)]\})} \quad (\text{to-getalls}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} 1 \\ \text{typecode} \\ nil \\ nil \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{array}\right]}{\Gamma, M, R \vdash \textbf{getall } \tau \textbf{ of } e \Rightarrow h(\{1, tc, nil, nil, \{\}, [\cup_{i=1}^{n}(\rho(e_i, \tau)])\})} \text{ (to-getallr}\tbinom{exp}{dyn}\text{)}$$

$$\frac{\Gamma, M, R \vdash e \Rightarrow l, R(l) = \lambda TO. \left[\begin{array}{l} \text{kind} \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_h, o_h)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, id = v, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_k] \end{array}\right], e' \Rightarrow v'}{\Gamma, R, M \vdash \textbf{annotate } e \textbf{ set } id \textbf{ to } e' \Rightarrow \text{ nop } \diamond R(l) \leftarrow \lambda TO. \left[\begin{array}{l} \text{kind} \\ \text{typecode} \\ [(s_1, o_1), \ldots, (s_h, o_h)] \\ \{c_1 = l_1, \ldots, c_n = l_n\} \\ \{a_1 = v_1, \ldots, id = v', \ldots, a_m = v_m\} \\ [e_1, \ldots, e_k] \end{array}\right]} \text{ (to-ann[s—r]}\tbinom{exp}{dyn}\text{)}$$

5.5.21 Query Like Operators

One of the main features of Manuzio is a set of operators with a syntax and semantics similar to those of SQL-like query languages operators. While such operators are generic and can work with any sequence of values, they are aimed to be the main way to interact with textual objects present in the persistent repository. The syntax and semantics of Manuzio's query like operators have its roots in object oriented database languages like the one presented in [Albano et al., 1985]. Query like operators are constructed to work on sequences of records in the form $[\{id = v_1\}, \ldots, \{id = v_n\}]$. The *in* operator is used to construct such sequences from both a normal sequence $[v_1, \ldots, v_n]$ and a repeated textual object with v_i as elements. The *in* operator is thus overloaded to work on both types. The rest of the operators works on these sequences regardless of their element's type.

Syntax

The syntax of query like operators should be familiar to readers used to relational databases. The *in* operators is used to construct sequence of records from sequence of values. Other operators works on such sequences to produce their results. The *where* operator is used to remove some sequence elements based on a conditional expression. *Each* and *some* returns a truth value indicating whenever all or some of the sequence elements conforms to some properties. The *select* operator performs a mapping from sequences to expressions, while the *groupby* operator organize the input sequence elements in sets basing the decision on one of the elements field value. Finally the *extend star* and *project star* operators performs the record extension and projection to all the elements of a sequence of records.

id **in** e (query-in)

e **where** e' (query-where)

each e **with** e' (query-each)

some e **with** e' (query-some)

select e **from** e' (query-select)

e **extend**$*$ e' (query-extend)

e **project**$*$ $\{id_1 : \tau_1, \ldots, id_n : \tau_n\}$ (query-project)

e **groupby** e' (query-groupby)

Static Semantics

Free Identifiers and Visibility Rules

The free identifiers of query operators are straightforward.

$$FV(id \textbf{ in } e) = FV(e) \qquad (\text{query-in}\tbinom{fv}{stat})$$

$$FV(e \textbf{ where } e') = FV(e) \cup FV(e') \qquad (\text{query-where}\tbinom{fv}{stat})$$

$$FV(\textbf{each } e \textbf{ with } e') = FV(e) \cup FV(e') \qquad (\text{query-each}\tbinom{fv}{stat})$$

$$FV(\textbf{some } e \textbf{ with } e') = FV(e) \cup FV(e') \qquad (\text{query-some}\tbinom{fv}{stat})$$

$$FV(\textbf{select } e \textbf{ from } e') = FV(e) \cup FV(e') \qquad (\text{query-select}\tbinom{fv}{stat})$$

$$FV(e \textbf{ extend}* e') = FV(e) \cup FV(e') \qquad (\text{query-extend}\tbinom{fv}{stat})$$

$$FV(e \textbf{ project}* \{id_1 : \tau_1, \ldots, id_n : \tau_n\}) = FV(e) \qquad (\text{query-project}\tbinom{fv}{stat})$$

$$FV(e \textbf{ groupby } e') = FV(e) \cup FV(e') \qquad (\text{query-groupby}\tbinom{fv}{stat})$$

Type Checking Rules

Most of the query operators works on sequences of records, built from sequences by the *in* operator invocation. For brevity the following abbreviation, used to denote the type of a record, holds for all the static rules in this section.

$$\tau_r = \{id_1 : \tau_1, \ldots, id_n : \tau_n\} \qquad (5.57)$$

The *in* operator will be defined in both rules query-ins$\binom{exp}{stat}$ and query-inr$\binom{exp}{stat}$ respectively for its application on sequences and textual objects.

$$\frac{\gamma, \varepsilon \vdash e : [\tau]}{\gamma, \varepsilon \vdash id \textbf{ in } e : [\{id : \tau\}]} \qquad (\text{query-ins}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : REP[\tau]}{\gamma, \varepsilon \vdash id \textbf{ in } e : [\{id : \tau\}]} \qquad (\text{query-inr}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau_r], e' : Bool}{\gamma, \varepsilon \vdash e \textbf{ where } e' : [\tau_r]} \qquad (\text{query-where}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau_r], e' : Bool}{\gamma, \varepsilon \vdash e \textbf{ each } e' : Bool} \quad (\text{query-each}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau_r], e' : Bool}{\gamma, \varepsilon \vdash e \textbf{ some } e' : Bool} \quad (\text{query-some}\tbinom{exp}{stat})$$

$$\frac{\gamma, \varepsilon \vdash e : [\tau_r], \gamma \cup \{id_i : \tau_i\}, \varepsilon \vdash e' : \tau' \ \forall \ i \in 1 \ldots n}{\gamma, \varepsilon \vdash \textbf{select } e' \textbf{ from } e : [\tau']} \quad (\text{query-select}\tbinom{exp}{stat})$$

Dynamic Semantics

As before the *in* operator will be presented in his two different implementations, when it is applied to sequences in query-ins$\binom{exp}{dyn}$ and when it is applied to repeated textual objects in query-inr$\binom{exp}{dyn}$.

$$\frac{\Gamma, M \vdash e \Rightarrow [v_1, \ldots, v_n]}{\Gamma, M \vdash id \textbf{ in } e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]} \quad (\text{query-ins}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow l, R(l) = \lambda TO. \begin{bmatrix} 1 \\ \text{typecode} \\ nil \\ nil \\ \{a_1 = v_1, \ldots, a_m = v_m\} \\ [e_1, \ldots, e_n] \end{bmatrix}}{\Gamma, M \vdash id \textbf{ in } e \Rightarrow [\{id = e_1\}, \ldots, \{id = e_n\}]} \quad (\text{query-inr}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e \Rightarrow rep[v_1, \ldots, v_n]}{\Gamma, M \vdash id \textbf{ in } e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]} \quad (\text{query-in}\tbinom{exp}{dyn}_{to})$$

$$\frac{\Gamma, M \vdash e' \Rightarrow b_i, e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]}{\Gamma, M \vdash e \textbf{ where } e' \Rightarrow [\{id = v_{k_1}\}, \ldots, \{id = v_{k_m}\}] \forall_{k_i} : b_i = \textbf{true}} \quad (\text{query-where}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e' \Rightarrow b_i, e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]}{\Gamma, M \vdash \textbf{each } e \textbf{ with } e' \Rightarrow \begin{cases} \textbf{true} & \text{if } b_i = \textbf{true } \forall i \in 1 \ldots n \\ \textbf{false} & \text{else} \end{cases}} \quad (\text{query-each}\tbinom{exp}{dyn})$$

$$\frac{\Gamma, M \vdash e' \Rightarrow b_i, e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]}{\Gamma, M \vdash \textbf{some}\ e\ \textbf{with}\ e' \Rightarrow \begin{cases} \textbf{false} & \text{if } b_i = \textbf{false}\ \forall i \in 1 \ldots n \\ \textbf{true} & \text{else} \end{cases}} \text{ (query-some}\tbinom{exp}{dyn}\text{)}$$

$$\frac{\Gamma, M \vdash e' \Rightarrow v'_i, e \Rightarrow [\{id = v_1\}, \ldots, \{id = v_n\}]}{\Gamma, M \vdash \textbf{select}\ e'\ \textbf{from}\ e \Rightarrow [v'_1, \ldots, v'_n]} \text{ (query-select}\tbinom{exp}{dyn}\text{)}$$

5.6 Conclusions

In this chapter the operational semantics of the Manuzio language has been given. For most of the language elements, only a minimal set of operators have been defined to maintain the specification short. In particular the sections about textual objects and query-like operators gives a formal explanations of the most peculiar features of Manuzio. In the next section an interpreter based on this specification is presented and discussed.

6

A Textual Database Prototype

"*More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.*" – William Allan Wulf

6.1 A General System Architecture

In this chapter the Manuzio system will be presented. The system allows users to store text in a persistent way and to analyze it through the use of the Manuzio language. While the development of a fully-featured system is beyond the scope of this thesis, we defined a set of design guidelines for a complete environment with the capabilities of dealing with users, annotations, queries, and corpus management.

1. The system should provide a way to store and to query textual corpora structured according to the Manuzio model. Since in the particular domain of application corpora tend to be large, it is important that the persistence layer is developed with an high regard for efficiency.

2. Each different corpus is contained in a textual database, which conforms to a specific Manuzio model. To manage the textual model evolution, the Manuzio language should have constructs to extend the model's schema with new types, to extend a type with new attributes and methods, both on single and repeated textual objects. Moreover, it must be possible to add or modify annotations on textual objects in a dynamic way.

3. The system should allow the access to concurrent users through an appropriate set of permissions. For instance, different groups of users could work with different sets of annotations on the same textual objects. The merge of different sets of annotations and a form of annotation sharing should be possible to allow collaborative analysis of texts.

4. The system should provide an user interface that allows users to express queries, write annotations, share results, and so on, in a natural and easy way. A graphical, user friendly, interface is planned for non-expert programmers

to perform assisted queries whose results are visualized with a choice of different graphical formats and mediums. The Manuzio programming language, instead, allows experienced users to express programs of arbitrary complexity through a set of high level, domain specific constructs.

5. To exchange texts and annotations with other systems the `XML` standard format will be used through a set of tools which facilitates the mapping between it and the Manuzio internal format. While `XML` has a set of already discussed drawbacks, it is the established format for data interchange and can be successfully used to export query results and portions of the corpus. In particular, `XML` can be considered the privileged way to load the initial textual data into the database, an operation which is done by a parsing process that can be automatic, semi-automatic or manual, depending on the complexity of the source data.

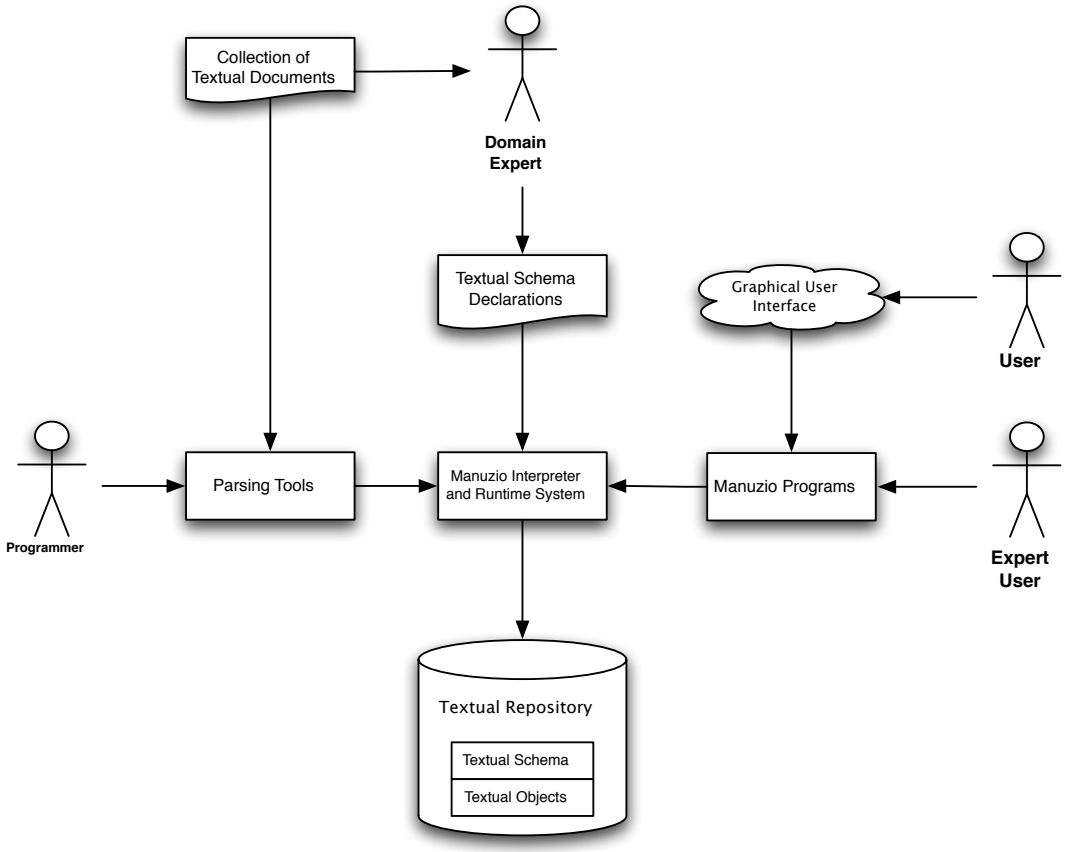

Figure 6.1: The Manuzio System Prototype.

In Figure 6.1 the workflow of the Manuzio system is shown. To create and interact with a textual repository the following steps must be observed:

- The encoder analyze the input text, and defines the textual object types to be used in the new textual repository. The definition of those types are then written as a textual schema declaration in the Manuzio language.

- A programmer writes a set of recognizer functions that, given the input text and the textual object types defined in the textual schema, identifies the instances of these textual object types in the text and returns a sequence of these objects. This set of functions is then used to create a parsing algorithm that populate the textual repository with textual objects by calling the textual objects creation primitives of the language.

- Different kind of users interact with the textual repository through either a graphical user interface or directly by writing Manuzio programs. Textual analysis programs written in Manuzio use a special command to connect to the textual repository and retrieve the contained textual object types directly from it. The data in the repository can then be queried by multiple users, and their results can be annotated and shared.

In the development of the first Manuzio system prototype the focus has been on the implementation of the language processor (the Manuzio interpreter) and the textual repository. The schema definitions, the corpus parsing, and the user interface aspects have been developed, instead, in a simpler way in order to have a working prototype. The resulting prototype let us evaluate critical Manuzio constructs early in its development and will influence future implementations.

The rest of the chapter is structured as follows: in Section 6.2 the Manuzio interpreter general structure is presented, at first from an high-level point of view, an then with a focus on its most relevant aspects. In Section 6.3, the possible implementation strategies for the persistent layer are discussed and the actual relational implementation is given. In Section 6.4, the other system current implementation's components are discussed. In Section 6.5 a case of study regarding Shakespeare's plays is shown, and, finally, in Section 6.6, some considerations on lessons learned from the implementation of the Manuzio language are given.

6.2 The Manuzio Interpreter

In this section the Manuzio interpreter implementation will be discusses. At first, in Section 6.2.1 an high level overview of the whole interpreter will be given. Later, the most important parts of the system will analyzed in more detail.

6.2.1 Interpreter Overview

The Manuzio interpreter is a software component that executes programs written in the Manuzio language. The current implementation of the Manuzio interpreter

has been written entirely in Ruby. Appendix A contains a brief introduction to the Ruby programming language in order to help the reader through the rest of the chapter examples and to introduce the object-oriented paradigm related terms that will be used through the interpreter definition.

The interpreter performs a classical read-evaluate-print loop, composed by the following steps:

- read an input sentence from the user;
- verify if the sentence is expressed in a valid Manuzio syntax;
- build an abstract syntax tree of the sentence;
- perform the type checking on the abstract syntax tree;
- if the types are correct, evaluate the sentence: the persistent layer and the textual database participates in the evaluation of textual object expressions;
- return the resulting value to the user.

These steps are summarized in Figure 6.2. Each of these steps can generate errors that must be cached and handled by the interpreter. The general behavior when an error occurs is to stop the execution of the current program and present a meaningful error report to the user in order to help her during the debug process.

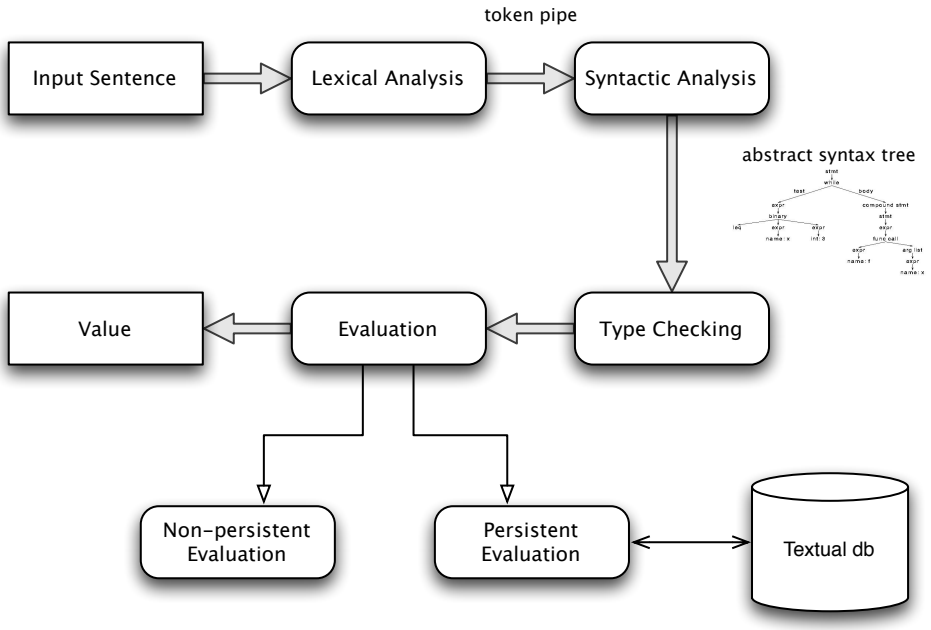

Figure 6.2: The interpreter steps.

The main task of the interpreter is the building and the evaluation of an Abstract Syntax Tree (`AST`). In our interpreter implementation each node of such tree contains an expression that represented by an object of an object-oriented, class based language. Each object class defines its instances behavior through a set of methods. The traditional procedures to type check and evaluate the language sentences are replaced by those specific behavior of the abstract syntax tree nodes. Each object is equipped with a *type* and a *value* methods, with the following signatures:

$$\text{type} : et \rightarrow (et, result)$$

$$\text{value} : ev \rightarrow (ev, result)$$

each of those methods takes in input an environment and returns a new, possibly modified, environment and a result. Since both are methods of an object, also the object instance variables are accessible. The result of type checking is a *language type*, while the result of evaluation is a *language value*. A language type is any type contained in the types environment γ, while a language value is a value taken from the constant environment Γ. Both language types and values are also represented, in the interpreter, by objects.

Types

Language types are represented by objects. Each predefined type or type constructor is represented by a different class. Predefined types, such as integers and booleans, that does not require parameters have no instance variables, while type constructors, like sequences or tuples, use instance variables to store their parameters. All the objects corresponding to types are equipped with a set of methods to perform different tasks required by the type checking algorithm. In particular each type has methods to:

- check the well formedness of the type. Simple types, like integers, are always well formed, while complex types, like tuples, can be bad formed if, for instance, have duplicate labels. This method is the implementation of the well-formedness rules for types defined in Chapter 5 by the $\binom{well}{stat}$ rules;
- check the subtype relation with another type. Each type defines its own subtyping rules through this method, as defined in Chapter 5 by the $\binom{sub}{stat}$ rules. For instance, integers are subtype only of themselves while tuples have a more complex behavior;
- a method to display the type to the user. This method can be implemented differently in different implementations, but in general it must return a string

containing a human-readable representation of the type or an encoded version of the type. For instance the integer types can be represented by the string "Int", or a tuple can be represented by an xml fragment that encodes its fields name and type.

Since our language include parametric polymorphism it is needed, for each type object, to include methods to:

- infer its free type variables, as defined by rule $\binom{ftv}{stat}$;
- to substitute type variables names with other names passed as parameters, as specified by the rule $\binom{sub}{stat}$.

The methods describes so far can be seen as an interface of the type, since all new types must implement, at least, these methods to work correctly with the interpreter. In Figure 6.3 the interface is shown in an uml style graphical notation, along with two example of conforming classes: the integer type, that does not have any instance variable, and the functional abstraction type constructor, that store in its instance variables information about its parameters and return type.

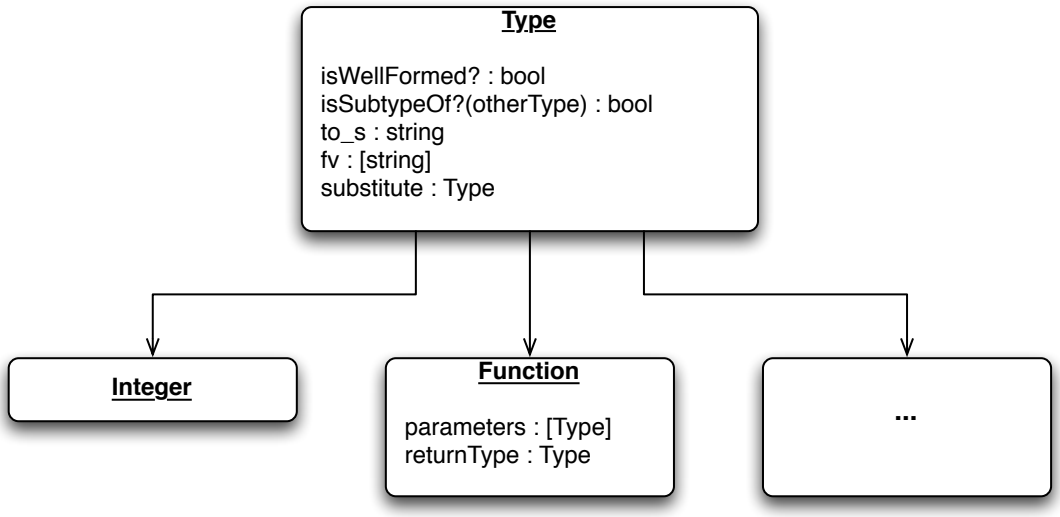

Figure 6.3: The type object interface and two examples of conforming objects, the integer type and functional abstraction type.

Values

In the Manuzio interpreter language values are represented as objects. Each object has one or more instance variables to store information about the represented value. For instance the object corresponding to a record has information about the values of its fields. All the classes corresponding to values conforms to a common interface that defines method to:

- check the equality of the value with another value passed as parameter. This evaluation must be carried out on the actual value and not referring to the object themselves.

- display the value in a meaningful way. As with types what is meaningful depends on the actual implementation. The most basic form of textual interpreters will require a string that holds a human readable representation of the value. The integer value of three could, for instance, be represented by the string "3".

In Figure 6.4 a fragment of the values hierarchy is shown with a graphical notation.

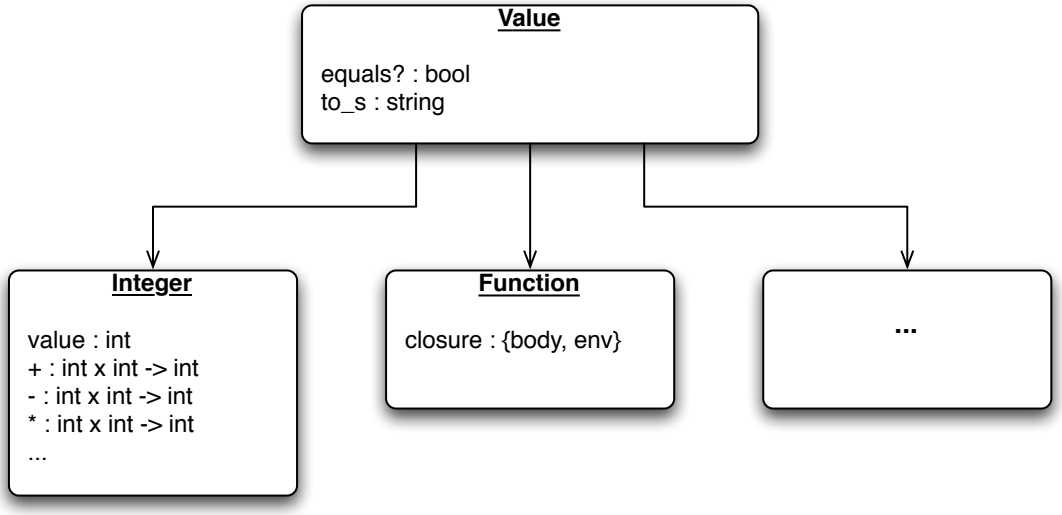

Figure 6.4: The value object interface and two examples of conforming objects, the integer value and functional abstraction value.

In addition, according to the value's type, other methods are defined to implement semantic operators on the values.

Environments and Memory

The Manuzio interpreter uses two environment: the first is the *types environment* and is used to type check the language expressions, the second, the *values environment*, is used to store the language identifier's values. Both are implemented as an object that incapsulates a modifiable sequence of pairs (identifier, type) (for the types environment) or (identifier, value) (for the values environment). Both the environments conforms to an interface that specify methods to:

- create an empty environment;

- clone the existing environment. This method creates a deep copy of the whole environment that can be passed as parameter to the typechecker or evaluator;
- add a new pair (identifier, type) (for types) or (identifier, value) (for values) to the environment;
- fetch the value or type associated with a given identifier;
- test for the presence of given identifier in the environment.

The *clone* method is important since each expression is typechecked or evaluated in a specific environment passed as a parameter to the expression object. Function bodies, for instance, are evaluated following the static scoping technique, so that the function closure contains a copy of the environment at the time of the function definition. This snapshot will be later extended with actual parameters values so that the current environment will be left untouched after the function call.

Since the Manuzio language can include the concept of variable, through the `Variable` bundle presented in Section 5.5.7, a form of memory is needed. The memory model of Manuzio is very simple. Each cell of memory is uniquely identified by an address and each cell can contain any value of the language, no matter how large. Differently from the environments, the memory is a global entity during the execution of a program and cannot be cloned. In the Manuzio interpreter the memory is a global object which has an interface that offers methods to initialize the memory, to provide a new memory address for a variable, and to fetch or write a value at a specific address.

Expressions

The Manuzio language is expression-based and each kind of expression is represented by a class, whose instances represent specific expressions. For example, the *if* statement have a correspondent class that can be instantiated with specific parameters (a conditional expression, a *then* expression, and an *else* expression) to obtain an object that represent the corresponding if expression. Classes that represent expressions are categorized in a hierarchy that follows the Manuzio language bundle structure.

Expression objects can have one or more instance variables to hold subexpressions. In the previous example, for instance, the *if* statement was composed of three different subexpressions. An expression is thus a recursive structure, composed by other expressions, that is isomorphic to the abstract syntax tree.

Each expression class conforms to an interface that specify three methods:

- a *type* method, that is called to perform the type checking;
- a *value* method, that is called to evaluate the expression value;

- a method to display the expression to the user. As with previous display method of types and values, this method depends on the actual implementation of the interpreter, but we can assume, without loss of generality, that this representation is simply a string.

Since the structure of expressions is recursive, their type checking, evaluation, and display methods are also recursive. To type check an *if* expression, for instance, the *type* method of all its three composing expressions must be called.

A special kind of expressions, called *polymorphic expressions*, is represented by a class with two or more subclasses. Each subclass represents an expression that shares its syntax with the expressions of the other subclasses, but differs for the type of its parameters. This behavior is in fact a form of *ad-hoc polymorphism*. During the type check a polymorphic expression has the task of identifying the expression that fits with the operand types among its children. If no such expression exists an error need to be signaled and the computation stops. If a match is found, however, the evaluation step will be carried out with the *value* method of the selected child. The *in* operator, for instance, works with the same syntax on both sequences and repeated textual objects. The type checker will inspect the type of the *in* operand and instruct the evaluator to use the right operator instance, in the case the operand's type is a sequence or a repetition, or returns a type error otherwise.

The interface of objects that represent expressions has, in addition, methods to specify their free identifiers and their definition identifiers, respectively called *fv* and *dv*. Each expression defines its own methods specified by the $\binom{fv}{stat}$ and $\binom{dv}{stat}$ rules.

In Figure 6.5 a fragment of the expressions hierarchy is shown. Round-corner boxes represent actual classes, while ovals are instances of polymorphic classes. In the figure, the `Expression` abstract class has three subclasses. Two are statements, representing the conditional expressions and the parenthesis expressions. The third class, the `DotExpression` is a polymorphic expression that specializes in the dot expression for records and objects.

6.2.2 Lexical Analyzer

The lexical analysis is the first part of the interpretation process that is performed by the *lexer*. Starting from a sequence of characters of the language *alphabet*, called *sentence*, the lexer main function is to partition that sentence in a sequence of *tokens*.

In the Manuzio Interpreter a class named `Token` is used to represent tokens. Each of its subclasses represents a specific type of tokens, like numbers, string literals, identifiers, and so on. Each `Token` subclass has a regular expression and a symbolic name as class variables. The regular expression is used to match the token with the input string. Instances of one `Token` subclass are used to represent specific tokens. Each one has a string that contains the characters of the alphabet that form the token, together with information about its location in the source code.

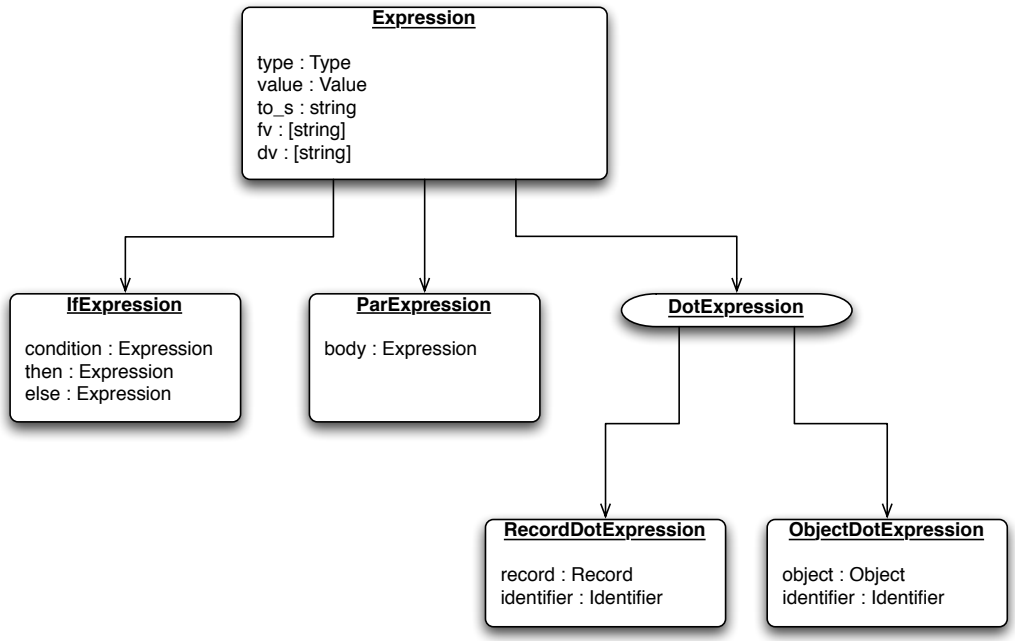

Figure 6.5: A fragment of the expressions class hierarchy.

The lexer works according to the code of algorithm 27. The input string is compared with the regular expression taken from one of the `Token` class subclasses. If a match with the head of the input is found then a token of that type has been recognized and is added to the output. If there is no such a match, then the lexer iterates over the rest of the tokens's regular expressions until it is finished. If at any time no match is found between the input and all the tokens, then a lexical error occurred: the computation is stopped and a lexical error is signaled. The lexer has also the task of cleaning its output of tokens, like blanks and comments, that must be ignored by the subsequent parsing process.

This trial-and-error behavior is an important concept that will be one of the main notions behind the Manuzio Interpreter extendibility feature. The lexer does not need to know in advance which tokens exists in the language, it just retrieve a list of them at execution time by asking to the main `Token` class a list of its subclasses[1]. The result is that the lexer's code does not need to be changed to accommodate new tokens, only a new subclass of the `Token` class has to be implemented.

If the lexical analyzer does not encounter any error its result is a data structure called *token pipe*. A token pipe is a sequence of tokens with an updateable index. The pipe is a simple array-like object that will not be discussed here. It is sufficient to say that it is equipped with methods to move the pointer (*next*), compare the pointed token with another one (*isa*), and saving/restoring the pointer position

[1]The order of such list is the reverse of the definition order. This way newly defined tokens are tested before old ones. This choice can be useful to overload part of the old tokens definition.

Source Code 27 The lexer algorithm.

```
def lex
    results = []
    while (!self.eos?)
      scan_result = nil
      scan_result = Token.each do |token_type|
                      scanned_substring = self.scan(token_type.regexp)
                      if scanned_substring
                        break token_type.new(scanned_substring, @current_line,
                                             self.pos)
                      end
                    end
      raise SyntaxError("Cannot recognize any token from: #{self.current}:
                        #{self.rest[0..20]}") unless scan_result
      results << scan_result
    end
    TokenPipe.new(results.reject {|e| (e.is_a? TokenEmpty) ||
                                       (e.is_a? TokenComment)})
  end
end
```

(*save*/*restore*).

6.2.3 Syntactic Analysis

When the lexical analysis terminates without errors the syntactic analyzer, also called *parser*, is executed in order to process the lexer output. The parser purpose is to check that the input sentence follows the language syntactic rules and to construct an abstract syntax tree that represent that sentence. Since the main requirement of the Manuzio parser is to be extensible and easy to understand, rather then efficient, the technique used to implement it is similar to a *recursive descent with backup* parsing[Aho et al., 1986].

The Manuzio parser is an object that is able to transform the lexer's token sequence into an abstract syntax tree. The parser makes use of a set of *components*, each one representing one particular expression of the language and with the knowledge necessary to build that particular expression subtree. Similarly to the lexer, the Manuzio parser has a trial-and-error behavior so that the main parser tries to build the `AST` by asking all the parser's subcomponents to do so until it gets a positive match from one of them or all of them are negative. In the first case the parser result is the `AST` returned by the component, while in the second case the parser will return an error to the user, since no components are able to parse the input token sequence. The process of building the `AST` is recursive, since each expression can be formed by subexpressions. The main parsing algorithm source code is shown in Source Code 28.

This decentralization of the parsing algorithm has the effect of isolating each production of the language grammar in a separate parsing component, thus in a separate object. The main parser assume the role of an immutable director that calls such components and collect their results in a sub-`AST`. While such isolation

could hinder performances and could make hard to perform many classical compiler optimizations, it makes, at the same time, the language parser modular, easy to understand, extend, modify, and, in our opinion, elegant.

Source Code 28 The main parsing algorithm.

```
#Parse all the input tokens until a full expression
#of the language is parsed a full expression is:
#1) until the end of the pipe
#2) until the separator token in found
def parseAll(pipe, separator=SEPARATOR)

  tmpExpression = nil
  stop = false
  while(!stop)
    stop = true
    @expParsers.each do |parser|
      break if pipe.isEmpty?
      break if (pipe.current == separator)
      parse_results = parser.parse(pipe, tmpExpression)
      if parse_results
        tmpExpression = parse_results
        stop = false
        break
      end
      #If the parser returns a result then something
      #has been found, we save the results in the
      #temporary parsed expression and we restart the
      #parsing process
    end
  end
  #when no parser can continue the parsing process, we
  #return the so-far-found expression
  return tmpExpression
end
```

Each parser component can represent of one of the following entities:

- one of the language types;
- one of the language expressions;
- a group of other parsing components;

In the Manuzio Interpreter each object that represent an expression or a type has a parser component counterpart, so that, for instance, there is a `ParseIfStatement` object to parse the *if* construct, a `ParseIntType` to parse the integer type, and so on. Each of those components has a *parse* method that reads the tokens from the pipe and checks if their sequence reflect the expression pattern. If so an `AST` of the expression is returned, else a null value is returned to indicate that the pipe does not contains that expression at his current position. In that case the token pipe index position is restored before returning. The *parse* method of components takes in input the token pipe and an expression. Such expression is the partial expression that has been recognized so far. Some productions can or can not be valid depending

on the existence of such partial expression (for instance infix operators requires a right side expression to be valid).

The main parsing algorithm described so far is implemented by a parser method called *parseAll*. Such method takes in input the pipe and a termination token. The termination token is used as a sentence separator when more then one sentence is inputted at the same time, and can be, for instance, a graphic token containing a semicolon. Moreover, the `parseAll` method can be used with a different termination token to stop the parsing at a certain keyword. For instance the *if* statement parse the condition by calling the `parseAll` method with the `THEN` token as termination token, so that the computation of the conditional expression stops when the "then" keyword is found. To implement a simple form of operators precedence the parser is also equipped with a *parseSingle* method. Such method apply only the first successful production of the grammar and then returns the `AST` as result. By using this method we can have a rudimental control on the shape of the resulting `AST` and thus control two levels of precedence among operators.

Groups, instead, are simply an aggregation of components that shares some peculiarities. Groups can contain sub-groups so that the parser components structure can be modeled as a tree. Infix operators, for instance, are considered a group which has, among its other components, subgroups to aggregate textual objects infix operators, arithmetic operators, and so on. The partition of components in group is not necessary to perform the parsing itself, but makes the resulting code more tidy, clean, and elegant. The main parser can call both a parser component directly or call a group of parsing components in exactly the same way.

A group behavior is similar to the one of the main parser: it calls the *parse* method of all its elements in turn to perform the parsing of the input pipe, and returns either a null value or the first valid `AST` it can build. We can see the structure of the parser as a tree, where the root is the main parser, the internal nodes are groups and the leaves are parsing components. By including or excluding parser components from the list of components of the main parser we can add or remove language elements without affecting other interpreter components. By grouping components we can implement easily the modularization required by the Manuzio language.

Keywords

One of the main idea behind conventional parsers is the one of *keywords*. A keyword is a reserved symbol, or group of symbols, that is used by the parser to identify some expressions, like the *if* keyword is used to identify the start of and `if` statement. Keywords must be known by the parser not only to perform the parsing itself, but also to prevent the user to use them as identifiers. Usually the set of a language keywords is small and hard-coded into the compiler or interpreter code, but with Manuzio, due to its extensibility features, it is not possible for the main parser component to know all of them in advance. The solution is again to distribute the

definition of keywords among the single parser components. Each parser component has a method *keywords* that returns a list of the keywords it requires. For instance the *if* expression parser component will require the keywords *if*, *then*, *else*, and *end*. By including such statement in the language means that those words cannot be used as identifiers anymore. Keywords of groups are the union of their components keywords. Since the structure of the parsers is a tree, the retrieve of keyword is done in a recursive way.

6.2.4 The read-evaluate-print cycle

After the abstract syntax tree of a program has been created, the interpreter performs the type checking of such tree by calling the *type* method of the tree root. If the check terminates correctly then the evaluation of the results is performed by calling the *value* method on the tree root. The interpreter carries out a read-evaluate-print to accept user input, evaluate the sentence, and display the results to the user. Results can either be a value or an error message. The most important part of such algorithm is show in Source Code 29.

Source Code 29 The main Interpreter cycle.

```
#perform initializations
loop do
        begin
                #read the user input line
                @pipe = LexicalAnalyzer::Lexer.new(line).lex
                expressions = Parser::Parser.instance.parse(@pipe)
                return nil unless expressions
                expressions.each do |exp|
                   type = exp.type(@et)
                   value = exp.value(@ev)
                   puts "#{exp} #=> #{value} : #{type}"
                end
        #On errors, prints what's wrong and ask for another line
        rescue Errors::SyntError, Errors::TypeClashError,
               Errors::SemanticError, Errors::DbConnectError,
               Errors::NullValueError => ex
                puts "#{ex.class}, #{ex.message}"
                retry
        end
end
```

While the interpreter should be able to read the input line from various sources and output the results in various formats, the current implementation is only able to perform a command line interaction or to read a program from an external file. Moreover, a set of commands that are not part of the Manuzio language are be implemented in the interpreter to:

- **saveenv**: save the current environment state persistently;
- **loadenv**: restore the saved environments;

- **printenv**: display the content of the environments;
- **clear**: clear the content of the environments;
- **printmem**: show the memory contents;
- **help**: display an help on interpreter commands;
- **quit**: quit the interpreter.

Such commands are useful for testing and debugging and are necessary in the testing environment required by an in-development language.

Error handling

All the interpreter steps can generate errors that must be caught and presented to the user in a meaningful way. Error handling is carried out through the raise of exceptions during the various steps of sentence evaluation. Such exceptions can be of different type and carry a string format message with them to provide useful information about the error to the user, so that corrections are made easier. The errors hierarchy and a brief description of their meaning is shown in Figure 6.6. Of particular importance the *semantic error* and the *type clash error* are the most used during debugging.

The Ruby's exception mechanism has been proven a valid way of handling errors without requiring any additional coding.

6.2.5 Conclusions

In this section we gave a set of informal guidelines for the development of an interpreter for the Manuzio language. The main goal is to build an highly modularized, easily extensible interpreter to be used in the process of language development and evaluation. For this reasons performances have not been considered a critical feature at this stage. In the next section the persistent features of the Manuzio's current implementation will be discussed.

6.3 Persistent Data

6.3.1 Textual Objects Persistency

The Manuzio language has persistent capabilities to handle textual objects. While other, more efficient, approaches could have been taken, to achieve a good tradeoff between speed and quick prototyping a relational database has been used to store persistent data. For an overview of other possible solutions see Section 6.6.

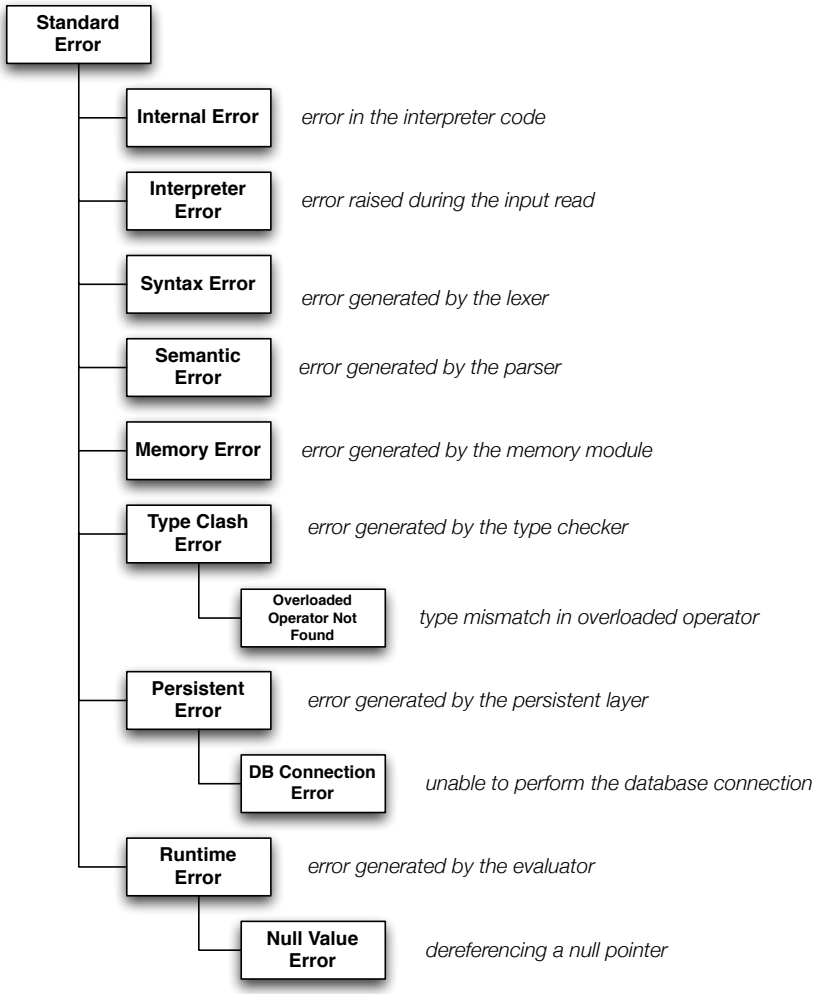

Figure 6.6: Error classes in Manuzio.

In this implementation textual objects structure and values are stored in a relational database. The Manuzio interpreter class that implements textual objects values performs the mapping between the database and the language through `SQL` queries. It is important to note that the language users have no perception of the underlying relational implementation, nor of the use of `SQL`. The persistent layer encapsulate the underlying implementation so that the user can deal with textual objects in the most natural way and without leaving the Manuzio language paradigm. Textual objects, like any other language value, are thus subject to typechecking, can be used as fields for records, can be passed as function parameters, and so on.
The definition of the Manuzio persistent layer followed the classical database design methodology: a conceptual schema for the Manuzio textual model has been designed first to give an high-level representation of textual objects.

The core of the conceptual schema, shown in Figure 6.7, is the `TextualObjects` object type. Such type represents the type of all textual objects, and is partitioned in two subtypes to represent single textual objects and repeated textual objects. Single textual objects participates in a component relation, described in Section 3.2, that is represented by a ternary association between single textual objects, textual objects, and a `Components` class that represent the label assigned to each component. In the schema we represented this association by interpreting each of its instances as an entity, called `ComponentObjects`, that has an $1 : n$ association with `Components`, `SingleTOs`, and `TextualObjects`. The association has an attribute to specify the relative number of a component textual object inside that specific component relation. Moreover, Single textual objects are associated with one or more textual chunks. A textual chunk is a pair of indexes in the form $(index, offset)$. Such pairs are used as inputs to query the `FullText` table and retrieve the underlying text of objects, through the *content* procedure. Repeated textual objects, represented by the `RepeatedTOs` class, have many single textual objects as elements.

Textual objects can have attributes, modeled by the `AttributeValues` class. Each attribute have a type that specify its data type and the label the attribute is denoted with. Each attribute value is associated with the `Attributes` class that stores all the previous values of the attribute together with other information about timestamps and users to form a sort of history of annotations on an object.

Each textual object has a type. Types, represented by the `Types` class, can have subtypes and are specialized in single and plural types. Single types are the types of single textual objects, while plural types are the types of repeated textual objects. Each single type has components of different type, while each plural type has elements of a certain type. Both can have attributes. The `AttributeTypes` class, together with `Types`, can be used to extrapolate the schema of the textual database.

Thanks to the persistent layer the user has the illusion that all textual objects, both single and repeated, are already instantiated when the connection to the database is performed and all other textual objects can be reached through the predefined constant *collection*. Under the hood, textual objects are stored in the

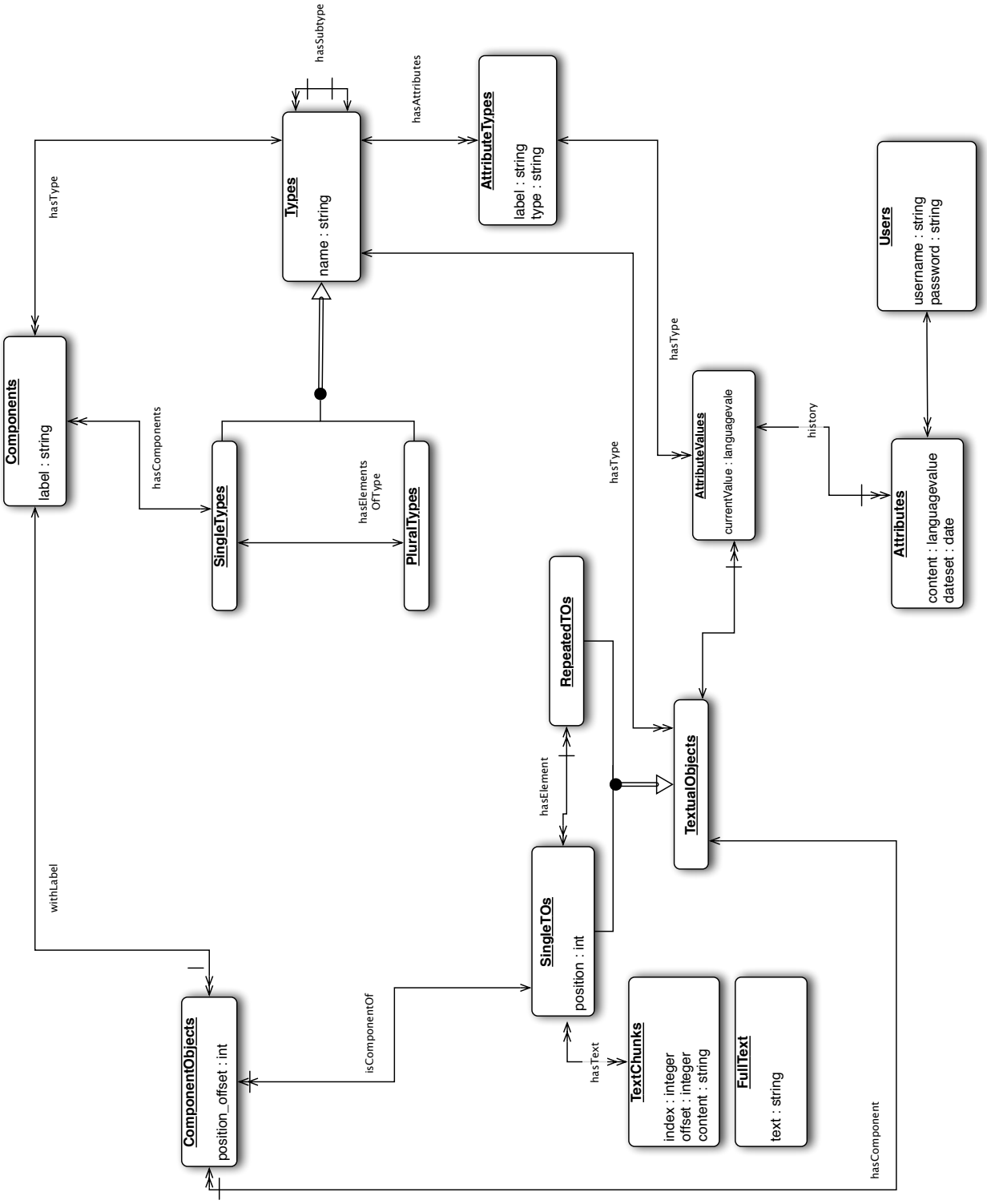

Figure 6.7: The Manuzio conceptual schema.

database and the language objects that represent textual objects holds only their unique identifier as an instance variable. When an operator is applied to a textual object value, it prepares and launches a query to the database and returns the result as a Manuzio value. For instance, if t is a textual object value, the command text of t launches a database query to retrieve the underlying text of t. The returned value will be a string, in the implementation's language sense, that will be packed in a Manuzio string value and returned as result.

In the current Manuzio Interpreter implementation textual object values are equipped with the following methods:

- **underling_text**(t): returns a Manuzio string value containing the underlying text of the textual object at address t.
- **components**(t, label): returns all the direct components of the textual object at address t that are labeled as the *label* parameter.
- **all_components**(t, T): returns all the components of the textual object at address t that are of type T, both directly and indirectly.
- **attributes**(t, label): returns the value of the attribute labeled as the passed parameter for the textual object at address t.
- **type**(t): returns the type of the textual object at address t.
- **parents**(t): returns a list of the parents of the textual object ad address t.
- **position**(t): returns the position of the textual object at address t. The position is computed following the objects lexicographic order, starting from the beginning of the full text.
- **position_in_type**(t, T): returns the position of the textual object of address t, following the lexicographic order, starting from the beginning of its parent of type T.

All these methods are computed by issuing a query to the database. Note that, while this set of methods is the minimal set that can be used to implement the textual object values as required by the language specification in Section 5.5.20, a more complex or numerous set of methods and queries can be used in order augment performances.

6.3.2 Database Relational Schema

The database schema contains not only information about the textual objects instances, but also about the Manuzio model being used, such as the names of the types and their relationships. This way the database schema is fixed and does not

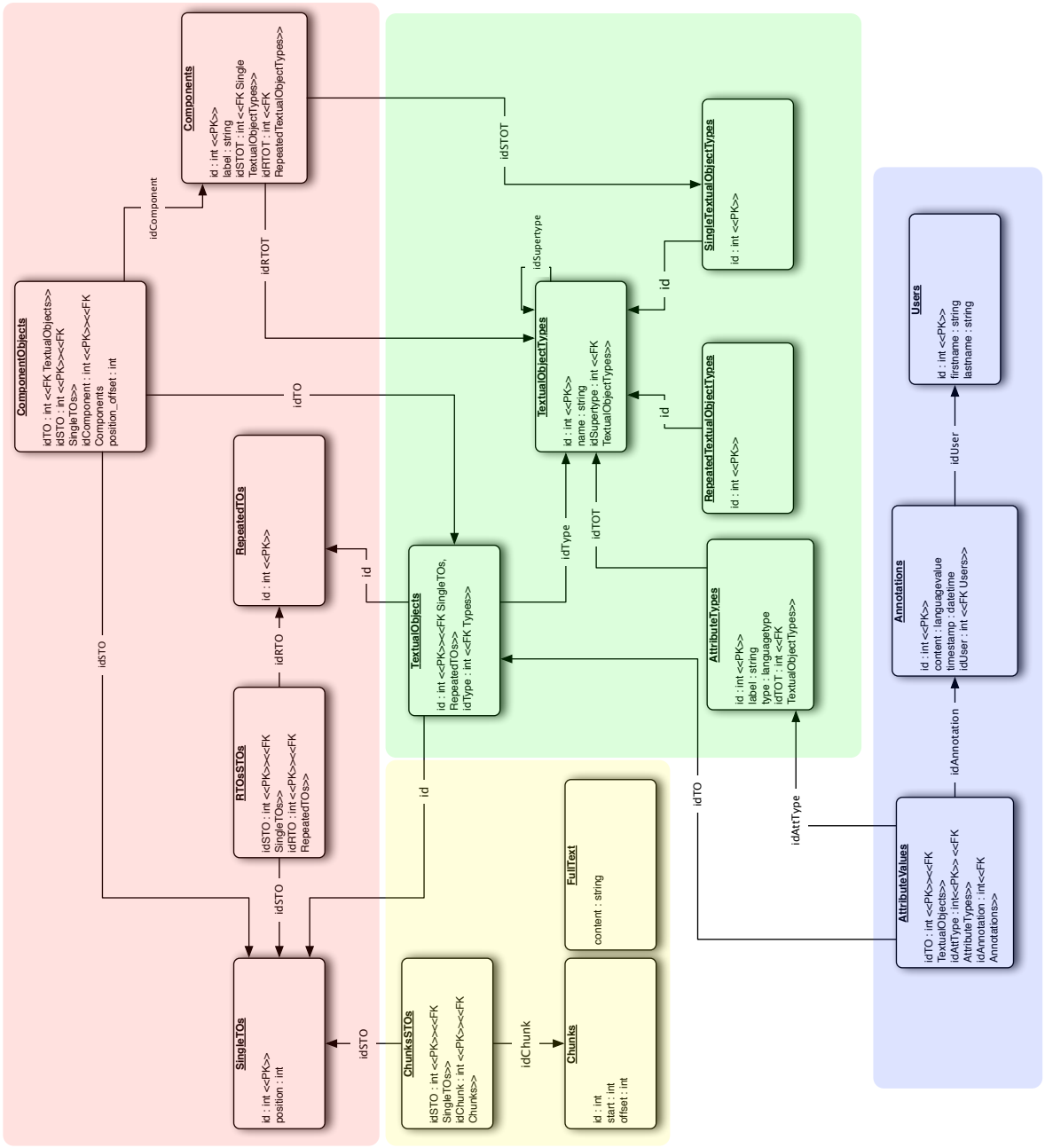

Figure 6.8: The RManuzio database relational schema.

need to be changed when the underlying model changes. The current relational implementation of the database is shown in Figure 6.8.

The database schema can be partitioned in four regions. The upper one, colored in red, contains the tables used to store textual object instances and the relation between them. In the mid left region, colored in yellow, the full text and the relations between text portions and textual objects are stored. The lower blue region contains the tables where the attributes of textual objects are saved, along with information about the users that made them. Finally, in the mid right, the green region contains the textual object types and other information about the textual model are stored.

Textual Objects Region

The textual object region is centered on the `TextualObjects` table, that specializes into single textual objects and repeated textual objects. Single textual objects are stored directly in the database, one row for each textual object present in the modeled text. Each single textual object is in relationship with one or more other singles through the `hasComponent` relation.

Repeated textual objects, instead, are not stored directly into the database due to their large cardinality. For this reason repetitions are, in general, created dynamically as an aggregation of single textual objects. The persistent storage of repeated textual objects is needed when such objects are annotated, so that a relationship with the annotations table can be created. From the language point of view, however, this process is completely transparent and have no impact on the user experience.

Textual Region

The textual region of the database contains the tables to store the full text and the underlying text of textual objects. While in the model we represented the full text as a big, monolithic string of characters, this solution does not works well in an actual implementation. The full text is thus segmented in smaller chunks stored in a separate table. Chunks are put in relationship with textual objects so that the underlying text of a textual object is the union of all its chunks text.

Annotations Region

The annotations region stores information about the annotations made on textual objects, the different users present in the database, and their relationships. Since in the current implementation the multiuser aspect and the dynamic annotation capabilities of the system are very limited, this region of the database is rather simple. While a real system with dynamic annotations and concurrent users should improve this part of the schema, in our single user static annotations prototype such simplicity is enough.

Since annotations can be of any Manuzio language type, a sort of encoding to store them in a relational table is needed. In our prototype a binary field has been used to store a string-encoded version of the stored value. Since the type of the annotation is known from the model, the stored string can be unpacked into the actual value without issues.

Model Region

The model region stores the names and the relationships of the textual types defined by the database's Manuzio model. Each type has a name and is in relation with other types that are its components with a certain label. Types can also have attributes, and each type is in relationship with its plural form. By navigating these tables the Manuzio language can, when connected to a database, reconstruct the whole Manuzio model used during the textual database creation, and know which types are required along with their structure.

6.3.3 Textual Objects Operators in SQL

The textual object values operators given in Section 6.3 work on the relational database by performing `SQL` queries. In this section the translation of these operators is given in source codes from 30 to 37.

Source Code 30 The *underlying_text*(t) `SQL` query for single textual objects.

```
1 SELECT   fulltext(C.start, C.offset)
2 FROM     SingleTOs as STO, ChunksSTOs as CT, Chunks as C
3 WHERE    STO.id = CT.idSTO and
4                  CT.idChunk = C.id and
5                  STO.id = t
```

Source Code 31 The *underlying_text*(t) `SQL` query for repeated textual objects.

```
1 SELECT   fulltext[C.start. C.offset]
2 FROM     SingleTOs as STO, ChunksSTOs as CT, Chunks as C, RepeatedTOs as RTO,
3                  RTOsSTOs as RTOSTO
4 WHERE    STO.id = CT.idSTO and
5                  CT.idChunk = C.id and
6                  RTO.id = t and
7                  RTO.id = RTOSTO.idRTO and
8                  RTOSTO.idSTO = STO.id
```

In Section 4.3 the concept of query composition in relation to lazy evaluation has been introduced. Without this technique the queries presented above are executed by the interpreter as soon as the relative language operator is encountered. For instance, the snippet, written in Manuzio, presented in Source Code 38 have poor performances in absence of lazy evaluation.

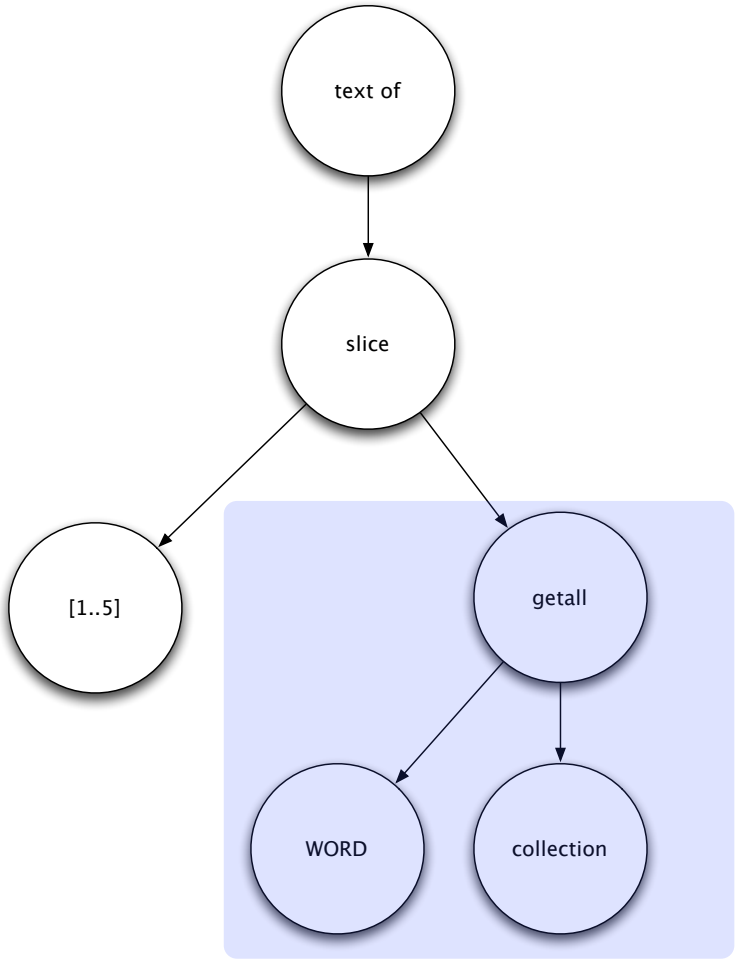

Figure 6.9: An abstract syntax tree fragment where query composition can be performed.

Source Code 32 The *components*$(t, label)$ SQL query.

```
SELECT  TO.id
FROM    SingleTOs as STO, ComponentObjects as CO, TextualObjects as TO,
                  Components as C
WHERE   STO.id = CO.idSTO and
                  TO.id = CO.idTO and
                  CO.idComponent = C.id and
                  C.label = label and
                  STO.id = t
```

Source Code 33 The *attribute*$(t, label)$ SQL query.

```
SELECT  A.content
FROM    TextualObjects as TO, AttributeValues as AV, Annotations as A,
                  AttributeTypes as AT
WHERE   AV.idTO = TO.id and
                  AV.idAnnotation = A.id and
                  AT.id = AV.idAttribute and
                  AT.label = label and
                  TO.id = t and
                  A.timestamp > tutti gli altri
```

In Figure 6.9 a simplified version of the code abstract syntax tree is shown. The highlighted part corresponds to the query execution. The language thus execute a query that returns the repeated textual objects of type *words* of the collection[2] by executing the query 32, only to subsequently slice it to just the first five of its elements with the repetition *slice* operator. Finally, the underlying text of such elements is required for displaying, so that the query 31 must be executed.

With lazy evaluation and query composition better performances can be achieved. When the language invokes a textual object operator the relative operator query do not get executed. The returned value is something similar to a functional language *thunk*, a non-evaluated value that represents a partial result of a computation. A thunk can be viewed as a function that takes no parameters and returns a value, in this case a repeated textual object, that gets executed only when such value is needed. Depending on subsequent usage of this value the computation can be carried on normally by executing the query, or the query itself can be modified to change its behavior.

The first query of the previous example, for instance, is shown in Source Code 39. With the query composition technique this query is prepared and stored in the returned thunk, but it's not executed right away. When the abstract syntax tree evaluation process encounter the *slice* node, the thunk is still an unevaluated value. A textual object thunk which value is needed can be:

- evaluated normally: if the thunk value is needed by a generic expression the normal behavior is, like in functional languages, to execute the thunk and use the resulting value;

[2]We assume here that the cardinality of such object elements is high.

Source Code 34 The *type*(*t*) `SQL` query.

```
SELECT  TOT.name
FROM    TextualObjects as TO, TextualObjectTypes as TOT
WHERE   TO.idType = TOT.id and
                TO.id = t
```

Source Code 35 The *parents*(*t*) `SQL` query.

```
SELECT  CONTAINER.id
FROM    TextualObjects as COMPONENT, ComponentObjects as CO,
                TextualObjects as CONTAINER
WHERE   COMPONENT.id = t and
                COMPONENT.id = CO.idTO and
                CONTAINER.id = CO.idSTO
```

- combined with the current operator: in this case the interpreter alters the query contained in the thunk to achieve the same results as the application of the operator to the thunk results. The resulting query is usually more efficient. The result of such combo is a new thunk containing the altered query.

For instance, in the previous example, the interpreter can alter the query to add the *slice* operator be incorporated in the previous query by using the `SQL` *limit ... offset* operator, as shown in Source Code 40.

In the final step of the computation the *text of* operator is applied to a thunk, so another combination can occur. The result is again a thunk that contains a modified query, shown in Source Code 41, and which return type is *String*. When the computation is ended the interpreter default behavior is to display the results to the user. At this moment the value of the thunk is needed and the interpreter has no other ways to modify the contained query, so it finally execute it and prints the results.

It is important to note that special considerations are needed when dealing with the textual object's annotation operators. These operators, in fact, carry the side-effect of changing the persistent value of an annotation in the textual repository in a manner that is similar to the update of a variable in memory. While a system where annotations are fixed, set when the textual database is created, can be viable for applications of text retrieval and simple analysis, annotations play, in our opinion, an important role. For this reasons, when dealing with lazy evaluation and query composition, the annotations operators must be taken in consideration. One possible solution, that will be inspected in future releases of Manuzio, could be the use of sessions, so that all the updates to the persistent storage are held until at least one unevaluated value is present. This behavior could be achieved, for instance, with sessions in a relational database.

While limited to certain specific combination of operators, the lazy evaluation and query composition technique can be used to achieve good performances on tasks that are typical for the domain of application, without the user leaving the elegant

Source Code 36 The *position*(t) SQL query.

```
SELECT  STO.position
FROM    SingleTOs as STO
WHERE   STO.id = t
```

Source Code 37 The *position_in_type*(t, T) SQL query.

```
SELECT  (COMPOMENT.position - CO.position_offset)
FROM    TextualObjectTypes as TOT, ComponentObjects as CO, SingleTOs as CONTAINER,
                TextualObjects as COMPOMENT
WHERE   COMPOMENT.id = t and
                COMPOMENT.id = CO.idTO and
                CONTAINER.id = CO.idSTO and
                CONTAINER.idType = TOT.id and
                TOT.name = T
```

programming language paradigm. The ideas behind query composition are, however, still in course of development and the Manuzio prototype does not, at the time of writing this document, include this experimental feature. We believe, however, that this approach, if carefully trimmed, could be of great help in the development of a more efficient Manuzio implementation.

6.4 Other Components

In this section the textual schema definitions, corpus parsing, and user interface of the Manuzio system will be overviewed. Due to time constraints these three components have been developed only in a simple way, so that the model and the language could be tested before undergoing a long, complex, implementation process. The discussion here will be brief, due to the fact that we just present the current implementation. A more in-depth discussion on the possible future implementations can be found in Section 5.6.

6.4.1 Textual Schema Definitions

Since the formal semantics of the Manuzio schema declarations is not completed yet, no parser exists for that part of the language. In the current implementation the textual object types to be inserted in a textual repository are encoded by hand in a data structure that, passed to the textual repository, fills the type-related tables of the underlying relational database.

In the current implementation of Manuzio the textual object types present in a textual database are specified by a set of declarations written directly in the Manuzio language. To perform this operation a new type constructor called *TEXTUALOBJECT* has been introduced to allow the specification of new textual object types directly from the language. This type constructor takes in input a tuple of compo-

Source Code 38 Underlying text of the first five words of the collection. This code is unoptimized.

```
1 > usedb "shakespeare"
2   #=> nop : Command
3 > text of (getall words of collection)[1..5]
4   #=> "Now fair Hippolyta our nuptial" : String
```

Source Code 39 Example of a component query translation.

```
1 SELECT  TO.id
2 FROM    SingleTOs as STO, ComponentObjects as CO, TextualObjects as TO,
3                 Components as C
4 WHERE   STO.id = CO.idSTO and
5                 TO.id = CO.idTO and
6                 CO.idComponent = C.id and
7                 C.label = "words" and
8                 STO.id = 0
```

nents, a tuple of attributes, the name of the associated repeated textual object type, and a tuple of attributes for such repetition. Note that this trivial way of defining schemas has been developed to have a working prototype to test the language capabilities in a controlled environment. In future implementations such types will be parsed, instead, from declarations like the ones in Source Code 42, presented in Section 6.5. In this way, schemas will be checked for consistency, so that the lattice-like required structure is enforced, and a set of other constraints will be present to allow to model special characteristics of the text[3]. It is matter of debate if this data definition language and its constraints should be given in a simple, yet limited, domain specific language, or rather through special declarations written directly in the Manuzio language, as shown in the examples. The domain specific language is simpler to use and read, but the constraint set must be limited. On the other hand, the use of a full programming language allows the definition of constraints of arbitrary complexity, but is also more difficult to use and to implement. A trade-off between the two solutions is currently being researched on, and will be discussed in Section 5.6.

6.4.2 Corpus Parsing

The corpus parsing process has the goal of instancing the textual database with the textual objects recognized in the text. In the current implementation of Manuzio this process has been done through an ad-hoc algorithm, different for each different corpus. The process is trivial, input `XML` documents are parsed through the Ruby programming language and the textual database is instanced with the recognized textual objects. When dealing with concurrent hierarchies the process is more com-

[3] An Haiku, for instance, is a form of Japanese poetry, consisting of 17 moras (or on), in three metrical phrases of 5, 7, and 5 moras respectively[Kobayashi and Lanoue, 1991].

Source Code 40 Example of a component query combined with a slice operator.

```
SELECT  TO.id
FROM    SingleTOs as STO, ComponentObjects as CO, TextualObjects as TO,
                Components as C
WHERE   STO.id = CO.idSTO and
                TO.id = CO.idTO and
                CO.idComponent = C.id and
                C.label = "words" and
                STO.id = 0
LIMIT 5 OFFSET 0
```

Source Code 41 Example of a component query combined with a slice operator and a text of operator.

```
SELECT  fulltext[C.start. C.offset]
FROM    SingleTOs as STO, ChunksSTOs as CT, Chunks as C, RepeatedTOs as RTO,
                RTOsSTOs as RTOSTO
WHERE   STO.id = CT.idSTO and
                CT.idChunk = C.id and
                RTO.id = RTOSTO.idRTO and
                RTOSTO.idSTO = STO.id and
                RTO.id IN (
                        SELECT  fulltext[CH.start. CH.offset]
                        FROM    SingleTOs as STO, ComponentObjects as CO,
                                        TextualObjects as TO, Components as C,
                                        SingleTOs as ELEMENTS, RTOsSTOs as RTOSTO
                                        ChunksSTOs as CT, Chunks as CH
                        WHERE   TO.id = RTOSTO.idRTO and
                                        RTOSTO.idSTO = ELEMENTS.idSTO and
                                        ELEMENTS.id = CT.idSTO and
                                        STO.id = CO.idSTO and
                                        TO.id = CO.idTO and
                                        CO.idComponent = C.id and
                                        C.label = "words" and
                                        STO.id = 0
                        LIMIT 5 OFFSET 0 )
```

plicated. Either the input is already encoded to accommodate such parallelism, or the parser will have to recognize itself non-encoded structures in some way. In future implementations a set of parsers are planned to allow a semi-automatic parsing of the most common encodings used in the humanistic field, like the `TEI`, standoff markup, `LMNL`, and so on. Due to the wide range of encodings and interpretations used by researchers, a fully automated process could be difficult to obtain, and each different source encoding system will have to be handled individually.

6.4.3 User Interface

The current implementation user interface is simply a command line interface that allows the user to interact with the Manuzio interpreter directly. Our goal is to develop an interface that allows the construction of a predefined family of queries through a graphical interface, like a classical form, but gives also, at the same time, the possibility to interact with the textual database through the Manuzio language

directly, so that non-predefined queries can be launched. The optimal ways to present and annotate query results in a simple yet coherent way across different models is still being researched on. A graphical web interface is currently being developed for a specific textual database. The results of this experiment will be used to develop a more general interface. Some considerations however, will be given in Section 5.6.

6.5 A Case of Study - Shakespeare's Plays

To show the potentiality of the Manuzio language in this section a small but realistic example will be presented. In our scenario we want to create a textual repository that store all of the Shakespeare's plays and perform some subsequent analysis on them. The source text we adopted, open source and freely available online, was a set of `XML` documents, conforming to a simple `DTD`, one for each of the plays. The first step to create a textual repository is to define which textual object types are present and their relationships. The decision is usually made by an expert of the domain of application that know which logical structures will be useful for the analysis and how they are related to each other. In our example the structure is rather simple, with only two parallel, unrelated structures to denote the metrical and prosodic structure of each speech. A special kind of speech, the epilogue, can be present and carries a title and subtitle as additional information. The Manuzio declarations for such structures are reported in code 42, while a graphical representation of them can be found in figure 6.10. These declarations are precessed by the Manuzio interpreter to instantiate a new textual repository. In the current implementation this means to create the appropriate tuples in the underlying relational database where information about textual object types are stored.

A parsing program, written either in Manuzio or in a language of choice, uses a set of recognizer functions to identify the textual objects of a certain type present in the text and return a list of them. Such objects are inserted into the textual repository through its primitive procedures for object creation. In the current example the parser has been written in Ruby; the Manuzio language has not been used mainly for the lack, at the time of writing, of an appropriate `XML` library. When the source text parsing is completed the textual repository is ready to be used.

From the Manuzio language the *usedb* command can be issued to connect to a textual repository. When the connection is made the textual object types present in the repository are loaded into the current environments. This means that all the types that has been declared during the creation of the textual repository are available during the analysis. Moreover, the *usedb* command instantiate a special variable *collection* that holds the singleton textual object of type *COLLECTION* and is used as an entry point for the textual data.

In Source Code 43 a newly-started Manuzio interpreter connects to the textual repository named `shakespeare`. As shown in the model, the textual object type

Source Code 42 Declarations in the Manuzio language for a simple Shakespeare's textual repository.

```
declare schema shakespeare
        type WORD = textualobjecttype
                attributes stem : Fun():String is get_stem(self.underlying_text)
                plural WORDS
        end

        type LINE = textualobjecttype
                components words : WORDS
                attributes meter : String
                plural LINES
        end

        type SENTENCE = textualobjecttype
                components words : WORDS
                plural SENTENCES
                plural attributes comment : String
        end

        type SPEECH = textualobjecttype
                components
                        sentences : SENTENCES,
                        lines : LINES
                attributes speaker : String
                plural SPEECHES
                plural attributes stagedirections : String
        end

        type EPILOGUE = textualobjecttype
                inherits SPEECH
                attributes title : {title : String, subtitle : String}
                plural EPILOGUES
        end

        type SCENE  = textualobjecttype
                components speeches : SPEECHES
                attributes title : String
                plural SCENES
        end

        type ACT = textualobjecttype
                components scenes : SCENES
                attributes title : String
                plural ACTS
        end

        type PLAY = textualobjecttype
                components acts : ACTS
                attributes
                        author : {firstname : String, lastname : String},
                        title : String,
                        kind : String
                plural PLAYS
        end

        type COLLECTION = textualobjecttype
                components plays : PLAYS
                attributes title : String, notes : String
        end
end
```

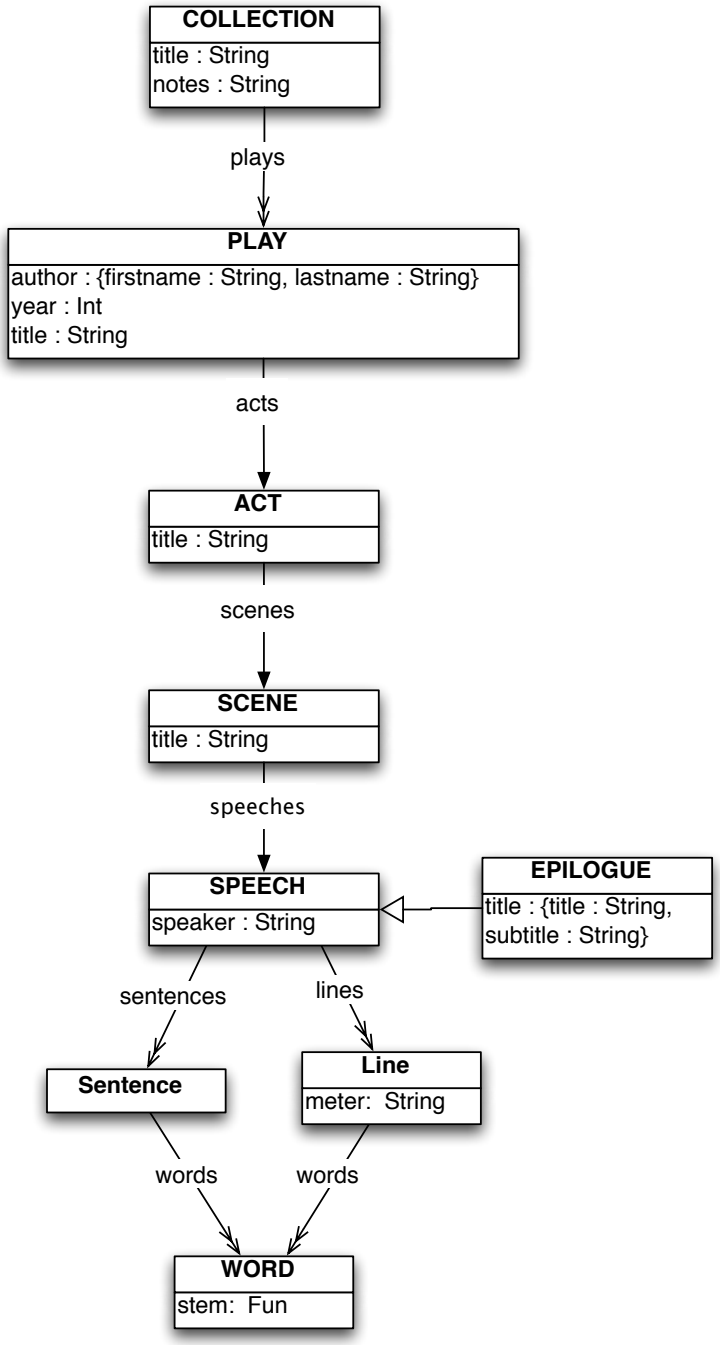

Figure 6.10: The Shakespeare's plays schema in graphical notation.

`COLLECTION` has an attribute `title` of type `String`. We can use the *get* operator to retrieve this attribute and check that the returned type is, in fact, a string value.

Source Code 43 Example 1: connection and retrieval of an attribute from a single textual object.

```
usedb "shakespeare"
get title of collection #=> "Shakespeare's plays" : String
```

In Source Code 44, instead, the *get* operator is used to retrieve other textual objects that are direct components of the collection. In our example the collection has only one component, a repeated textual object of type `PLAYS` called `plays`. We can assign the retrieved object to a variable to use it in subsequent expressions. The *get* operator can work also on repeated textual objects to retrieve the attributes of their elements, acting as a mapping; in the code the titles of all plays are retrieved: the result is a sequence of strings.

Source Code 44 Example 2: retrieval of components from a single textual object and attributes from a repetition.

```
let plays = get plays of collection
get title of plays
  #=> ["The Tragedy of Antony and Cleopatra",
  #=>  "All's Well That Ends Well", ...] : [String]
```

The *get* operator can be applied to repetitions also to retrieve components: its semantics is again similar to a mapping. In Source Code 45 the acts of all the plays are retrieved. Note that, in this case, the result is not a sequence of repeated textual objects as it would be in a real mapping, but an implicit flattening of the results is performed so that the result is again a repeated textual object of type `ACTS`. This expression's results are, in fact, equivalent to the retrieval of all the acts of the collection. Since `ACT` is not a component of `COLLECTION`, such task must be performed by the *getall* operator, that takes in input a textual object t and a textual object type T and returns all the textual objects of type T that are direct or indirect components of t.

Source Code 45 Example 3: retrieval of components from a repetition.

```
get acts of plays
getall ACT of collection
```

The underlying text of a textual object can be retrieved with the *text of* operator. The result is a string both when the operator is applied to single textual objects and when it is applied to repeated textual objects. In the latter case an implicit flattening of the results is performed; the underlying text of all the repetition's elements are

concatenated in a single string (with a predefined separator). An example of both cases is presented in Source Code 46.

Source Code 46 Example 4: retrieval of underlying text from single and repeated textual objects.

```
1 text of head plays #=> "Nay, but this dotage of our general's ...
2                    #=> High order in this great solemnity." : String
3 text of plays #=> "Nay, but this dotage of our general's ...
4               #=>  We were dissever'd: hastily lead away." : String
```

More complex queries can be expressed through the query-like operators of Manuzio. In Source Code 47, for instance, we show how to retrieve the title of all plays by means of these operators. In Source Code 48, instead, we make use of the *where* operator to reject all the plays that do not have the *kind* attribute set to the vale "Tragedy". The returned value is a sequence of strings that contains the titles of all the Shakespeare's tragedies.

Source Code 47 Example 5: retrieve the title of all plays.

```
1 select title of p
2 from p in get plays of collection
```

Source Code 48 Example 6: retrieve the title of all tragedies.

```
1 select title of p
2 from p in get plays of collection
3 where get kind of p = "Tragedy"
```

In Source Code 49 we return a repeated textual object composed by all the plays that have at least one word with the underlying text equals to "king" by using the *some ... in* operator. The same logic applied to the example in Source Code 50, where we search the entire collection for sentences where all the words are very short.

Source Code 49 Example 7: retrieve the title of all the plays that contains the word "king".

```
1 select title of p
2 from p in get plays of collection
3 where some w in (getall WORD of p) with text of w = "king"
```

Source Code 50 Example 8: retrieve sentences composed by only short words.

```
1 let short_sent = select s
2   from s in getall SENTENCE of collection
3   where each w in (getall WORD of s) with size of w < 5
```

Repeated textual objects can also have their own annotations, that are different from the annotations on their elements. We can annotate every textual object, like the sentences retrieved by the previous example, with the *annotate ... set ... to* operator. In the example shown in Source Code 51, we then retrieve the annotation with the *get* operator. Note that, since the `SENTENCES` type has an annotation named `comment`, this annotation gets retrieved instead of collecting the annotations of the elements.

Source Code 51 Example 9: annotation of a repetition and retrieval of annotations on repetitions.

```
1 annotate short_sent set comment to "this sounds funny!"
2 get comment of short_sent #=> "this sounds funny!" : String
```

Finally, in program Source Code 52 an example of textual analysis program is shown. The first instruction stores in the constant `play` the play with title "A Midsummer Night's Dream". Then all the speeches which contains at least a word with stem "love" of that play are saved in `loveSpeeches`. The percentile of such speeches in the whole play is computed and presented to the user. The last selection iterates over the `loveSpeeches` repetition and returns a record with two fields, the first containing the `speaker` as a string, the second containing the number `n` of speeches with the stem "love" spoken by that speaker as in integer. Finally the top ten results are presented to the user. The results are reported in Source Code 53.

Source Code 52 Example 10: compute the top ten characters by the number of speeches spoken which contains the word stem "love".

```
1  usedb "shakespeare";
2
3  let play =
4      p in (get plays of collection)
5      where p.title = "A Midsummer Night's Dream";
6  let loveSpeeches =
7      s in (getall Speech of play)
8      where some w in (getall Word of s) with (get stem of w) = "love";
9  let r = (size of loveSpeeches)/(size of getall Speech of p) * 100;
10
11 output "There is a " + r + "% of love speeches.";
12 let love_speech_count_by_speaker =
13     select {speaker = s.speaker, n=(size of s.partition)}
14     from s in (speeches groupby speaker)
15     order by n;
16 output "The top 10 love spekaers are:";
17 output love_speech_count_by_speaker[1..10];
```

Source Code 53 Example 10 results.

```
There is a 18.4% of love speeches.
The top 10 love speakers are:
[{speaker="LYSANDER", n=17},
 {speaker="OBERON", n=13},
 {speaker="HERMIA", n=12},
 {speaker="DEMETRIUS", n=10},
 {speaker="TITANIA", n=9},
 {speaker="HELENA", n=9},
 {speaker="THESEUS", n=6},
 {speaker="Thisbe", n=3},
 {speaker="PUCK", n=3},
 {speaker="QUINCE", n=3}]
```

6.6 Evaluation of the Language

In the development of Manuzio the modular design has been of great use to dynamically refine various features and operators of the language. The process of implementing and using the language has, however, also put in evidence some shortcomings. The lessons learned from the implementation of the language are presented in this section.

- **Syntax**: the language syntax can still be considered heavy, with little syntactic sugar to aid the developers in their task. The main reasons behind this drawback was the need to keep the language parser simple and concentrate on other, more important, aspects like the language semantics. To make the language more appealing to the eye, however, a more advanced parser could accept a simpler syntax. Also, in the current implementation, the syntax of operators has been given mostly by keywords like *distance ... from ...*, *getall ... of ...*, or *parent ... of ...*; this is in contrast with the choices made by most general-purpose programming languages, where the majority of operators are denoted by non-alphabetic symbols. We feel, however, that in the specific domain of application our approach could be successful provided that some refining work is done at syntax level.

- **Standard Libraries**: although the lack of standard libraries is not a real deficiency of the language itself, a little more work is needed to define which operators should be implemented as native and which are to be implemented as library components. Good standard libraries have often been at the base of a language success or failure, so we feel that Manuzio should feature a large set of predefined functions on textual objects and strings, comprising, among others, of specific textual functions such as phonetic algorithms, computation of textual distances, and so on.

- **Programming Paradigm**: the choice of using a functional language has been taken to exploit the mathematical elegance of such approach, as well as

the idea that a literary text is a kind of data that requires few or no updates. For this reason, the use of lazy evaluation and memoization can be useful to augment performances in presence of persistently stored data. The importance of dynamic annotations as a feature of primary importance, however, required some additional efforts, since their presence introduces side-effects and breaks the elegance of a pure functional approach. It is matter of debate if the next version of the Manuzio language will still be a functional language or if a radical departure from the functional programming paradigm should be taken.

- **Potentiality of the Language**: during our internal tests the language has proven useful to solve many problems of literary analysis. The seamless integration between the programming language and the query language makes easy for the developers to write textual analysis programs; without the need of an external query language, the coding experience feels more natural and smooth. The ability of the Manuzio type system to catch type errors early, at compile time, is of great use in a persistent environment where retrieval operations can be very time-demanding.

- **Textual Objects Operators**: in the current implementation the number of textual objects operators has been kept to a minimum to test the language features in a simple and controlled environment. During our tests, however, the need of some other operators arose to simplify some recurrent tasks. The dependent product operator, for instance, would be a useful addition to express queries like "for all the lines with at least a word of 13 characters, return a tuple containing that line and the corresponding word". An example of such behavior is given in Source Code 54 by the (sill unimplemented) *times** operator.

Source Code 54 Example of dependent product between the lines and the words of a play *p*.

```
1 select {line = text of l, word = text of w}
2 from l in (getall LINE of p) times* w in (getall WORD of p)
3 where size of w = 13
4 #=> [{line = "for, as it is a heartbreaking to see a handsome man",
5 #       word = "heartbreaking"},
6 #       ...,
7 #     {line = "Tell him thy entertainment: look, thou say",
8 #       word = "entertainment"}]
```

Conclusions and Future Work

In this work we presented a model, a language, and a system to store and query textual data.

The Manuzio model describes the features of textual corpora in a precise and flexible way through the use of a formal language. Differently from others widespread models, the Manuzio data model is in effect a directed acyclic graph where multiple, concurrent textual structures can be represented effectively without recurring to workarounds. The model is based on the idea that text can be seen as a multiple hierarchy of objects; each object represents a recognizable portion of the text with some logical meaning like a word, a paragraph, and so on. Each textual object has a type that specify that object interface, and objects can be in a component relation or their types can be in a subtype relation. By modeling a text like a hierarchy of objects, we have been able to reuse some of the object-oriented concepts successfully to define a complex model that is, at the same time, sound and easily understandable.

We introduced the abstract concept of textual database as a persistent repository of textual objects that represent a specific collection of texts. A Manuzio model is specified through a set of definitions written in a formal language, each document of a given textual database shares the same model. This is a debatable point: from a certain point of view our model commits to the "fallacy of prescience" that afflict also markup models[Liu and Smith, 2008]; we assume that the text has a structure and that the encoder has a prior knowledge of that structure. Other models allow to specify schemas where the structure is present but not enforced, so that relationships between objects can be explored during the analysis process. This approach is a good choice for some applications like, for instance, text criticism. Our choice to use a somewhat more rigid model derives from the context where this model is born (the analysis of texts with a precise and solid structure) and from the choice to have a powerful programming language with static type checking as our query language. Since, in our opinion, annotations are an important part of the research process on corpora, Manuzio features a full set of types and type constructors that can be used to define annotations on objects. From a language point of view, annotations are just values of the language: they can be of any type allowed by the language, carry complex, structured information, and are subject to type checking. Moreover, the in-development Manuzio system is planned to allow different layers of annotations on the same textual databases with a form of annotation sharing, history, and merging to create a rich research and discussion environment.

The main result of the thesis is, in our opinion, the development of the Manuzio language as a language to interact with and query textual repositories. The Manuzio

language has been designed to be a functional, interpreted language with static type checking. The language main goal is to be a full programming language, but have a set of predefined, high level operators to allow users with limited programming experience to use it as a query language for textual databases. To achieve this, it has been developed as an interactive language, so that every step of computation produces a partial result that the user can review, annotate, and so on. The static type checking ensure that the operations performed are type consistent, catching the majority of errors that the average Manuzio users commit before the actual execution instructions. The Manuzio language has two peculiarities that make it distinct from other, general-purpose, programming languages: the modular design and the set of high-level operators to interact with persistent textual databases. The modular design allows the language to be flexible: it is easy to remove or modify functionalities from it as well as adding new ones. We found this design effective at this stage of prototyping, since it allows to play with different language's features in an easy, natural way. We chose to follow this path also to be able to develop the specific language characteristics in a free environment, without having to perform hard decisions a priori.

Manuzio features a set of high-level operators to interact with textual databases. While not a fully persistent language, Manuzio treats textual objects like persistent values, that are transparently read and written into the textual database without any additional effort from the user. Textual objects are, in fact, native values of the language, so that they can be manipulated, passed as parameters, and are type checked as any other value. We designed the language to be able to work with persistent textual data without the user experiencing the paradigm mismatch that is found, for instance, when performing `SQL` queries from traditional programming languages. In the current prototype the support for persistence is limited: it is possible to connect to only one database at the time and the disconnect command is not implemented yet. In future implementations multiple, namespaced database connections will be possible. The testing has proven, however, that the interaction with textual objects is done in a natural way and that type checking, when applied to textual objects, helps avoiding an elevated number of errors when writing commands. A set of query-like operators are provided to help the user to express queries on textual material[4] in a familiar, `SQL`-like, way. Unlike `SQL`, however, each of the query keyword in in fact a different operator that can be used singularly. In our tests we found that, despite the somewhat rigid syntax typical of prototypal languages, Manuzio is suitable to develop textual analysis programs, mostly due to the seamless integration of the programming and query language. We are confident that a future version of Manuzio, with a more elegant and clear syntax and superior performances, will be a valuable tool in the field of humanities research.

The Manuzio language is meant to be used in a system where users can share annotations on texts. Time constraints, however, reduced the full system specification

[4]The query-like operators, however, work also on other data types.

to a simple sketch. The language interpreter and the textual repository have been fully developed, even if not optimally, while other aspects, like the creation of the textual database and the graphical interface, have been assumed and implemented only in a simple way to have a working prototype and test the feasibility of the project. The Manuzio interpreter has been developed using the Ruby programming language, a dynamically typed, object-oriented, interpreted language with a rich set of libraries. Thanks to the fast-prototyping features of Ruby the implementation of the interpreter has been pleasant and the resulting code is easy to understand and modify. Efficiency considerations have been made during the implementation of the Manuzio interpreter, but at the current state of work only a few basic optimizations have been implemented. The goal of this thesis is not to have a fast, solid system to use for complex textual analysis problems, but to develop a prototype that can prove the effectiveness of our model and can be used, in a controlled environment, to test the intuitiveness of the language query-like constructs.

The textual repository has been implemented using a relational technology. Again, the choice has been made to exploit the well-established strengths of such model, but a more efficient, ad-hoc solution could help performances. In particular, it is important to note that even a medium-size corpus analyzed at word-level can comprise millions of single textual objects. Repeated textual objects, which cardinality is comparable to the power set of the single textual objects set, must also be represented in some way to be able to be annotated. In our implementation, for instance, such repetitions are stored in the database only if they are annotated.

The achievements reached in this work represent a base for the development of a complete, efficient, system based on the Manuzio language. To reach such a goal, however, a some additional work is needed. The major missing points of our current implementation are listed below, with hints for solution, as future work that will follow the redaction of this thesis.

- Formal specification and parser for the Manuzio language's textual object declarations. In the current implementation textual schemas are defined by declarations in the Manuzio language, but the textual object types to be inserted in the repository during the parsing process must be encoded by hand in a data structure rather than be parsed directly from such declarations. The schema structure should be also checked for consistency and the specification of constraints and methods written directly in the Manuzio programming language should be possible.

- A new, efficient, implementation of the language interpreter and persistence layer. A possible approach to improve both the performances and the acceptance of our approach could be to add the textual object operators and the persistence layer to a wide-spread programming language by extending its interpreter. To reach such goal, the Scala programming language [Odersky et al., 2004] seems to be the right choice of a functional, type-checked, and extensible language based on the wide-spread Java architecture. Another possible approach

could be to shift the textual object access primitives to stored procedures in a relational database system, and access them from different programming languages by developing a simple library for each of such languages. This approach is currently being tested with a prototype written with the PostgreSQL relational database's procedural languages [Douglas and Douglas, 2003] and the Ruby programming language [Matsumoto and Ishituka, 2002]. A more efficient, ad-hoc, storage solution could also be researched to specifically store textual objects.

- Design of a multi-user environment. In the current implementation concurrency have been taken into consideration, but the system is currently single-user. The textual repository should provide primitives for concurrent access to textual data so that multiple users can query the text and add annotations to their results or share them.

- Design a series of semi-standard parsing algorithms. While existing digital editions of literary text are encoded in many different, often non-standard, ways, the design of algorithms to parse source texts encoded in `XML` (possibly enriched with the workarounds to represent concurrent hierarchies), `LMNL`, or other markup languages could make the process of parsing the source text easier.

- Graphical representation of the results. Since the data model of Manuzio is dynamic, different textual repositories can be constituted by different hierarchies, have a different unit type, and so on. Such freedom, however, means also that a generalized graphical presentation of query results is not easy to achieve. While ad-hoc, schema-dependent, solutions can be easily realized, a more general approach could make easier to develop textual analysis programs with a rich graphical user interface.

A

The Ruby Programming Language

Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves. – Yukihiro Matsumoto

A.1 Introduction to Ruby

In this section the basics of the Ruby programming language are given in order to allow the reader to fully understand the Manuzio interpreter code fragments shown in Section 6.2.

Ruby is a recent programming language that blends parts of Perl, Smalltalk, Eiffel, Ada, and Lisp to form a new language. The main goal of Ruby is to be "natural" rather then "simple". Ruby is a general purpose programming language with support for multiple programming paradigms, including functional, object-oriented, imperative, and reflective. Ruby implement a dynamic type system and automatic memory management, and it is considered, for this reasons, a good language to write clean and short code fast.

In Ruby, everything is an object. Every bit of information and code can be given their own properties and actions. Object-oriented programming calls properties by the name instance variables and actions are known as methods. Rubys pure object-oriented approach is most commonly demonstrated by the code fragment 55 which applies an action to a number.

Source Code 55 In Ruby, everything is an object.

```
1 5.times { print "Ruby is good for fast prototyping!" }
```

In many languages, numbers and other primitive types are not objects. Ruby follows the influence of the Smalltalk language by giving methods and instance variables to

all of its types. This eases ones use of Ruby, since rules applying to objects apply to all of Ruby.

Ruby type system follows a style o dynamic typing called *duck typing*. In duck typing an object's current set of methods and properties determines if that object is valid in a particular context, rather then its name or its position in a hierarchy of classes. With this technique the programmer is concerned with just those aspects of an object that are actually used in the code, rather then with the type of the object itself. Dynamic typing may result in runtime errors and is thus often considered an error-prone technique that makes bugs difficult to locate in code. Dynamic typing, however, allows a more dynamic programming environment, faster compiling, and typically makes easier and effective the use of metaprogramming. A form of static type checking for Ruby also exists [Furr et al., 2009].

While Ruby often uses very limited punctuation and usually prefers English keywords, some punctuation is used to decorate Ruby. Ruby needs no variable declarations. The type of a variable is inferred by its value and is dynamic, so it can be changed by subsequent assignments. To denote the scope of variables, instead, a simple naming convention is used:

- *var* is a local variable;
- *@var* is an instance variable;
- *$var* is a global variable.

These symbols enhance readability by allowing the programmer to easily identify the roles of each variable. It also becomes optional to use the *self* keyword to denote instance variables of an object.

Ruby is a flexible language, since it allows the programmers to freely alter any of its parts. Every part of ruby can be removed or redefined, down to the most basic types, thanks to its "everything is an object" philosophy.

Source Code 56 Example of class re-opening in Ruby.

```
1 class Numeric
2   def plus(x)
3     self.+(x)
4   end
5 end
6
7 y = 5.plus 6
8 # y is now equal to 11
```

For instance, in Source Code 56, the `Numeric` class is extended at runtime to include a new way of computing the sum of two numbers. The example Source Code 56 also shows other two important characteristics of the language: classes can be re-opened at any time to be modified, and operators are only a syntactic sugar for object

methods. This high degree of flexibility allows programmers to write code in a very natural way, so that the creative process of designing the code is not hindered by language technicalities.

Ruby objects support single inheritance only, but a sort of multiple inheritance can be achieved through the use of *mixins*. A mixin is performed between a class and a module (a collection of namespaced methods). When such operation is performed the class receive all the module methods. For example, any class which implements the each method can mixin the Enumerable module, which adds a set of methods that use each for looping.

Source Code 57 Example of module inclusion in Ruby.

```
1  module PhoneticAlgorithms
2    def methaphone(s)
3      ...
4    end
5
6    def double_methaphone(s)
7      ...
8    end
9
10   def soundex(s)
11     ...
12   end
13 end
14
15 class TextualObject
16   include PhoneticAlgorithms
17
18   def metaphone
19         PhoneticAlgorithms::metaphone(self.text)
20   end
21
22   ...
23 end
```

The Source Code 57 shows how a set of phonetic algorithms can be grouped in a module so that they can be easily added to a class representing textual objects in a subsequent time.

Multiple inheritance is a powerful mechanism, but is also considered too complex and sometimes restricting. Mixins, on the other hand, are a more natural way to share common behaviors between classes. Modules can be also seen as a form of *interface*. In the rest of the section we will refer to the interface of a class as to a collection of methods, with signatures, that the class must posses to be valid. While Ruby is an highly dynamic language that lacks a native concept of interface, other techniques can be used to achieve a similar behavior [Tate, 2006].

Another feature that makes Ruby natural to use is the extensive use of blocks. A programmer can attach a closure to any method, describing how that method should act. The closure is called a block and has become one of the most popular features for newcomers to Ruby from other imperative languages.

Source Code 58 Example of Ruby blocks usage.

```
1 author_websites =
2   ["Plautus", "Seneca", "Terence"].map do |author|
3     "http://www." + author.downcase + ".com"
4   end
```

In Source Code 58, the block is described inside the *do ... end* construct. The map method applies the block to the provided list of authors, and an array of ideal authors website addresses is returned.

The Ruby programming language has strong reflection capabilities that allow the use of metaprogramming techniques. With this feature Ruby programs can alter themselves at runtime, inspect objects types and interfaces, and so on. An important characteristic of Ruby metaprogramming is the presence of *hook methods*. An hook is a spacial method that gets called when a specific event occurs. For instance, the `inherited` hook method is an empty class method that gets called whenever a class is inherited by another class. Both classes are passed as parameters, and the user can overload this method freely to create custom behaviors.

Source Code 59 Usage of reflection and hook methods in Ruby.

```
1  #A class is a good father is it remembers the name of its children in the
2  #@children array.
3  module GoodFather
4
5    #When the module is included in a class, add the following
6    #methods to the class
7    def self.included(klass)
8      klass.instance_eval do
9        @children ||= []
10
11       #When inherited, add the new child to the children array, head insert
12       def inherited(child)
13         @children.insert(0, child)
14       end
15
16       #iterates through the children
17       def each(&block)
18         @children.each {|c| yield c}
19       end
20     end
21   end
22 end
```

For instance, the `GoodFather` module defined in Source Code 59 can be included in any class to maintain a class variable with a list of all the classes that inherited from that class.

This introduction to the Ruby programming language, while short and far from complete, should be enough for readers with programming experienced to understand the implementation code fragments shown in the next section. More information on Ruby can be found in [Flanagan and Matsumoto, 2008, Tate, 2006,

Thomas and Hunt, 2000].

List of Figures

List of Sources

Bibliography

[Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A theory of objects*. Springer Berlin.

[Abrams and Harpham, 2008] Abrams, M. and Harpham, G. (2008). *A glossary of literary terms*. Wadsworth Pub Co.

[Aho et al., 1986] Aho, A., Sethi, R., and Ullman, J. (1986). Compilers: principles, techniques, and tools. *Reading, MA,*.

[Al-Khalifa et al., 2003] Al-Khalifa, S., Yu, C., and Jagadish, H. V. (2003). Querying structured text in an xml database. In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *SIGMOD Conference*, pages 4–15. ACM.

[Albano et al., 1985] Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260.

[Albano et al., 1995] Albano, A., Ghelli, G., and Orsini, R. (1995). Fibonacci: A programming language for object databases. *The VLDB Journal The International Journal on Very Large Data Bases*, 4(3):403–444.

[Albano and Procopio, 1998] Albano, A. and Procopio, F. (1998). Realizzatione di un interprete ad oggetti per un linguaggio funzionale polimorfico. *rapporto tecnico, Università di Pisa, Dipartimento di Informatica*.

[Allen, 1991] Allen, J. (1991). Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6(4):341–355.

[Alschuler, 1995] Alschuler, L. (1995). *ABCD... SGML*. International Thomson Computer Press.

[Atkinsonf et al., 1990] Atkinsonf, M., Bailey, P., Chisholmt, K., Cockshottf, P., and Morrison, R. (1990). An approach to persistent programming. *Readings in Object-Oriented Database Systems*.

[Barnard et al., 1988] Barnard, D., Hayter, R., Karababa, M., Logan, G., and McFadden, J. (1988). SGML-based markup for literary texts: Two problems and some solutions. *Computers and the Humanities*, 22(4):265–276.

[Berrie, 2000] Berrie, P. (2000). Just in time markup for electronic editions. *NewWorlds of Learning*.

[Bradley and Short, 2005] Bradley, J. and Short, H. (2005). Texts into databases: the evolving field of new-style prosopography. *Literary and linguistic computing*, 20:3.

[Bray et al., 2000] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and Yergeau, F. (2000). Extensible markup language (XML) 1.0. *W3C recommendation*, 6.

[Bruce, 2002] Bruce, K. (2002). *Foundations of object-oriented languages: types and semantics*. The MIT Press.

[Bruno and Murisasco, 2006] Bruno, E. and Murisasco, E. (2006). Describing and querying hierarchical xml structures defined over the same textual data. In Bulterman, D. C. A. and Brailsford, D. F., editors, *ACM Symposium on Document Engineering*, pages 147–154. ACM.

[Burnard and Sperberg-McQueen, 2005] Burnard, L. and Sperberg-McQueen, C. (2005). Il manuale tei lite. *Introduzione alla codifica elettronica dei testi letterari. Milano: Edizioni Sylvestre Bonnard*.

[Buzzetti, 2002] Buzzetti, D. (2002). Digital representation and the text model. pages 461–88.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On Understanding types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522.

[Carletta et al., 2003] Carletta, J., Evert, S., Heid, U., Kilgour, J., Robertson, J., and Voormann, H. (2003). The NITE XML Toolkit: flexible annotation for multimodal language data. *Behavior Research Methods, Instruments, and Computers*, 35(3). Special issue on Measuring Behavior.

[Carlson, 1967] Carlson, G. (1967). Literary works in machine-readable form. *Computers and the Humanities*, 1(3):75–102.

[Chamberlin, 2002] Chamberlin, D. (2002). XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615.

[Connolly, 1997] Connolly, D. (1997). Xml: Principles, tools, and techniques. *WWW J*, 2.

[Coombs et al., 1987a] Coombs, J. H., Renear, A. H., and DeRose, S. J. (1987a). Markup systems and the future of scholarly text processing. *Commun. ACM*, 30(11):933–947.

[Coombs et al., 1987b] Coombs, J. H., Renear, A. H., and DeRose, S. J. (1987b). Markup systems and the future of scholary text processing. *Commun. ACM*, 30(11):933–947.

[Deerwester et al., 1992] Deerwester, S. C., Waclena, K., and LaMar, M. (1992). A textual object management system. In Belkin, N. J., Ingwersen, P., and Pejtersen, A. M., editors, *SIGIR*, pages 126–139. ACM.

[DeRose et al., 1997] DeRose, S., Durand, D., Mylonas, E., and Renear, A. (1997). What is text, really? *ACM SIGDOC Asterisk Journal of Computer Documentation*, 21(3):1–24.

[DeRose, 1997] DeRose, S. J. (1997). Further context for what is text, really. *SIGDOC Asterisk J. Comput. Doc.*, 21(3):40–44.

[DeRose, 2004] DeRose, S. J. (2004). Markup overlap: A review and a horse. In *Extreme Markup Languages*.

[DeStefani, 1999] DeStefani, A. (1999). Un interprete estendibile ad oggetti per linguaggi funzionali polimorfici. *Tesi di Laurea, Università di Venezia, Dipartimento di Informatica*.

[Dipper, 2005] Dipper, S. (2005). Xml-based stand-off representation and exploitation of multi-level linguistic annotation. In Eckstein, R. and Tolksdorf, R., editors, *Berliner XML Tage*, pages 39–50.

[Doedens, 1994] Doedens, C. (1994). Text databases: one database model and several retrieval languages. *Computational Linguistics*, 24(2).

[Douglas and Douglas, 2003] Douglas, K. and Douglas, S. (2003). *PostgreSQL*. Citeseer.

[Durusau and O'Donnell, 2002] Durusau, P. and O'Donnell, M. (2002). Concurrent markup for xml documents.

[Fernandez, 2004] Fernandez, M. (2004). *Programming Languages and Operational Semantics: An Introduction*. King's College Publications.

[Flanagan and Matsumoto, 2008] Flanagan, D. and Matsumoto, Y. (2008). *The ruby programming language*. O'Reilly Media, Inc.

[Fraser et al., 1986] Fraser, C. A., Barnard, D. T., and Logan, G. M. (1986). Generalized markup for literary analysis. In *SIGDOC '86: Proceedings of the 5th annual international conference on Systems documentation*, pages 27–31, New York, NY, USA. ACM.

[Friedl, 2006] Friedl, J. (2006). *Mastering regular expressions*. O'Reilly Media, Inc.

[Furr et al., 2009] Furr, M., An, J., Foster, J., and Hicks, M. (2009). Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866. ACM New York, NY, USA.

[Goldfarb and Rubinsky, 1990] Goldfarb, C. and Rubinsky, Y. (1990). *The SGML handbook*. Oxford University Press, USA.

[Hilbert et al., 2005] Hilbert, M., Schonefeld, O., and Witt, A. (2005). Making CONCUR work. In *Extreme Markup Languages*, volume 2005. Citeseer.

[Hockey, 2004] Hockey, S. (2004). Electronic Texts in the Humanities. Principles and Practice. *Literary and Linguistic Computing*, 19(2).

[Hosking and Chen, 1999] Hosking, A. and Chen, J. (1999). Persistent Modula 3: An Orthogonally Persistent Systems Programming Language-Design. Implementation, Performance. In Proceedings of the 25th International Conference on Very Large Data Bases (Edinburgh, Scotland.

[Huitfeldt and Sperberg-McQueen, 2001] Huitfeldt, C. and Sperberg-McQueen, C. (2001). TexMECS: An experimental markup meta-language for complex documents. *URL http://www. hit. uib. no/claus/mlcd/papers/texmecs. html*.

[Iacob and Dekhtyar, 2005a] Iacob, I. and Dekhtyar, A. (2005a). Processing xml documents with overlapping hierarchies. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, page 409. ACM.

[Iacob and Dekhtyar, 2005b] Iacob, I. and Dekhtyar, A. (2005b). Towards a query language for multihierarchical xml: Revisiting xpath. In *Proceedings of the 8th International Workshop on the Web and Databases (WebDB 2005), Baltimore, Maryland, USA*. Citeseer.

[Iacob et al., 2004] Iacob, I., Dekhtyar, A., and Zhao, W. (2004). Xpath extension for querying concurrent xml markup. Technical report, Citeseer.

[Ide, 1994] Ide, N. (1994). Encoding standards for large text resources: The text encoding initiative. In *COLING*, pages 574–578.

[Jagadish et al., 2004] Jagadish, H. V., Lakshmanan, L. V. S., Scannapieco, M., Srivastava, D., and Wiwatwattana, N. (2004). Colorful xml: One hierarchy isn't enough. In Weikum, G., König, A. C., and Deßloch, S., editors, *SIGMOD Conference*, pages 251–262. ACM.

[Kelley, 1995] Kelley, D. (1995). *Automata and Formal Languages: an introduction*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Klarlund et al., 2003] Klarlund, N., Schwentick, T., and Suciu, D. (2003). Xml: Model, schemas, types, logics, and queries. In Chomicki, J., van der Meyden, R., and Saake, G., editors, *Logics for Emerging Applications of Databases*, pages 1–41. Springer.

[Kobayashi and Lanoue, 1991] Kobayashi, I. and Lanoue, D. (1991). *Issa, cup-of-tea poems: selected haiku of Kobayashi Issa*. Asian Humanities Pr.

[Lee and Chu, 2000] Lee, D. and Chu, W. (2000). Comparative analysis of six xml schema languages. *ACM Sigmod Record*, 29(3):76–87.

[Light, 1997] Light, R. (1997). *presenting XML*. Sams Indianapolis, IN, USA.

[Liu and Smith, 2008] Liu, Y. and Smith, J. (2008). A relational database model for text encoding. *Digital Studies/Le champ numérique*, 3(3).

[Loeffen, 1994] Loeffen, A. (1994). Text databases: A survey of text models and systems. *SIGMOD Record*, 23(1):97–106.

[Mastandrea, 1992] Mastandrea, P. (1992). De fine versus, repertorio di clausole ricorrenti nella poesia dattilica latina.

[Matsumoto and Ishituka, 2002] Matsumoto, Y. and Ishituka, K. (2002). *The ruby programming language*. Addison Wesley Publishing Company.

[Maurizio and Orsini, 2008] Maurizio, M. and Orsini, R. (2008). A model and query language for literary texts.

[McEnery, 2003] McEnery, T. (2003). Corpus linguistics. In Mitkov, R., editor, *The Oxford Handbook of Computational Linguistics*, Oxford Handbooks in Linguistics, pages 448–463. Oxford University Press.

[Meijer et al., 2006] Meijer, E., Beckman, B., and Bierman, G. (2006). Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA. ACM.

[Mller and Strube, 2003] Mller, C. and Strube, M. (2003). Multi-level annotation in mmax.

[Morrison et al., 1999] Morrison, R., Connor, R., Cutts, Q., Kirby, G., Munro, D., and Atkinson, M. (1999). The napier88 persistent programming language and environment. *Fully Integrated Data Environments*, pages 98–154.

[Müller, 2005] Müller, C. (2005). A flexible stand-off data model with query language for multi-level annotation. In *ACL*. The Association for Computer Linguistics.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the scala programming language. *LAMP-EPFL*.

[Petersen, 1999] Petersen, U. (1999). The extended mdf model.

[Petersen, 2002] Petersen, U. (2002). The standard mdf model. *Unpublished article. Obtainable from URL: http://emdros.org*.

[Petersen, 2004a] Petersen, U. (2004a). Emdros: a text database engine for analyzed or annotated text. In *COLING '04: Proceedings of the 20th international conference on Computational Linguistics*, page 1190, Morristown, NJ, USA. Association for Computational Linguistics.

[Petersen, 2004b] Petersen, U. (2004b). Mql programmer's guide.

[Petersen, 2004c] Petersen, U. (2004c). Mql user's guide.

[Pierce, 2002] Pierce, B. (2002). *Types and programming languages*. The MIT Press.

[Savinov, 2005a] Savinov, A. (2005a). Concept-oriented model. *Encyclopedia of Database Technologies and Applications*.

[Savinov, 2005b] Savinov, A. (2005b). Concept-oriented programming. *Encyclopedia of Information Science and Technology*.

[Savinov, 2008] Savinov, A. (2008). Concepts and Concept-Oriented Programming. *Journal of Object Technology*, 7(3):91–106.

[Schreibman et al., 2004] Schreibman, S., Siemens, R., and Unsworth, J. (2004). *A companion to digital humanities*. Blackwell.

[Scott, 1999] Scott, M. (1999). *Programming language pragmatics*. Morgan Kaufmann Pub.

[Selinger, 2001] Selinger, P. (2001). Lecture notes on the lambda calculus. *University of Ottawa, Fall*.

[Sperberg-McQueen et al., 1994] Sperberg-McQueen, C., Burnard, L., and Bauman, S. (1994). Guidelines for electronic text encoding and interchange.

[Sperberg-McQueen and Huitfeldt, 2004] Sperberg-McQueen, C. and Huitfeldt, C. (2004). Goddag: A data structure for overlapping hierarchies. *Lecture Notes in Computer Science*, pages 139–160.

[Steele, 1990] Steele, G. (1990). *Common LISP: the language*. Digital Pr.

[Tate, 2006] Tate, B. (2006). *From Java to Ruby: things every manager should know*. Pragmatic Bookshelf.

[Tennison and Piez, 2002] Tennison, J. and Piez, W. (2002). The layered markup and annotation language (lmnl). In *Extreme Markup Languages®*.

[Thomas and Hunt, 2000] Thomas, D. and Hunt, A. (2000). *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Travis and Waldt, 1995] Travis, B. and Waldt, D. (1995). *The SGML implementation guide*. Springer Berlin etc.

[Wasserman and Sherertz, 1976] Wasserman, A. and Sherertz, D. (1976). A balanced view of MUMPS. *ACM SIGPLAN Notices*, 11(4):16–26.

[Witt, 2004a] Witt, A. (2004a). Multiple hierarchies: new aspects of an old solution. In *Proceedings of Extreme Markup Languages*. Citeseer.

[Witt, 2004b] Witt, A. (2004b). Multiple hierarchies: new aspects of an old solution. In *Extreme Markup Languages®*.

[Zanella, 2008] Zanella, S. (2008). Un modello ad oggetti per il trattamento di testi e sua applicazione ad alcuni problemi di analisi linguistica. In *Tesi Specialistica in Informatica per Discipline Umanistiche*.

[Zi et al., 1992] Zi, D., Waclena, K., and Spackman, S. (1992). Using a Lazy Functional Language for Textual Information Retrieval.

Index