TALLINN TECHNICAL UNIVERSITY

Combination of Object-Oriented and Logic Paradigms

Tarmo Uustalu

A Thesis submitted to the Tallinn Technical University
in partial fulfillment of the requirements
for the degree of Master of Engineering

Written at the Dept. of Computer Systems and Telematics,
The Norwegian Institute of Technology (NTH), Trondheim,
Apr – Aug 1991

Supervisors: Prof. Reidar Conradi and Prof. Jan Komorowski

Combination of Object-Oriented and Logic Paradigms

Tarmo Uustalu

Abstract

In the present thesis, possibilities to combine the object-oriented (OO) and the logic software developing paradigms have been studied. A number of existing proposals and solutions has been reviewed and categorized. A proposal has also been made towards a new solution which tries to capture the essence of OO in terms of modal logic, and which could be implemented as a modal logic programming system.

Within the proposal, a logic $\mathrm{MU}_{\mathcal{U}}$, and its advanced variants $\mathrm{MU}'_{\mathcal{U}}$ and $\mathrm{MU}''_{\mathcal{U}}$, allowing different varieties of inheritance modes, have been designed and studied. All three are meant for reasoning about a static object hierarchy with static objects. Dynamics has been treated in a logic $2\mathrm{MU}_{\mathcal{U}}$, and in its corresponding advanced variants. Their design proceeded from a belief that in OO essentially two orthogonal but similar dimensions of evolution are involved—object hierarhy and time. Values of dynamic attributes can be understood as inherited along the (usually linear) hierarchy of time instants. The pragmatics of the resolution calculi of the mentioned logics has been discussed on the basis of examples.

Keywords: *object oriented paradigm, logic paradigm, modal logic, resolution calculi, modes of inheritance, dynamic states.*

Objektorienteeritud ja loogilise paradigma kombineerimine

Tarmo Uustalu

Resümee

Käesolevas diplomitöös on uuritud võimalusi objektorienteeritud (OO) ja loogilise tarkvaraparadigma kombineerimiseks. On kirjeldatud ja klassifitseeritud olemasolevaid ettepanekuid ja lahendusi. On ka tehtud ettepanek uue lahenduse suunas, kus OO mõisteid ja tehnikaid on püütud haarata modaalse loogika vahenditega ning mida saaks realiseerida modaalloogilise programmeerimise süsteemina.

Ettepaneku raames on välja töötatud loogika $\mathrm{MU}_{\mathcal{U}}$ ning tema erinevaid pärimise mooduseid võimaldavad arendatud variandid $\mathrm{MU}'_{\mathcal{U}}$ ja $\mathrm{MU}''_{\mathcal{U}}$. Kõik kolm on mõeldud deduktsiooniks staatiliste objektidega staatilise objektide hierarhia üle. Dünaamikat käsitavad loogika $2\mathrm{MU}_{\mathcal{U}}$ ning tema vastavad edasiarendused. Nende väljatöötlus lähtus veendumusest, et OOsse on sisuliselt kätketud kaks ortogonaalset, kuid sarnast evolutsioonidimensiooni—objektide hierarhia ja aeg. Dünaamiliste atribuutide väärtusi võib vaadelda pärituina üle (tavaliselt lineaarse) ajahetkede hierarhia. Nimetatud loogikate resolutsiooniarvutuste pragmaatikat on arutatud näidetele tuginedes.

Võtmesõnad: *objektorienteeritud paradigma, loogiline paradigma, modaalne loogika, resolutsiooniarvutused, pärimise moodused, dünaamilised olekud.*

Acknowledgements

This thesis was written at Institutt for datateknikk og telematikk (IDT), Norges tekniske høgskole (NTH), while holding a scholarship of Internasjonal avdeling, NTH. To the staff of both I owe many thanks.

I appreciate very much the trouble that my supervisors—professors Reidar Conradi and Jan Komorowski—had to see with me. Besides the numerous remarks on the contents of the work, the general guidance and the encouragement that I received from both, Reidar Conradi significantly improved the language of the thesis, and Jan Komorowski helped me with many sources of literature. I am grateful to Reidar Conradi also for the wonderful trip to West Norway which he arranged, and which I could join, and to Jan Komorowski also for the wonderful bicycle which he lent me, and which served me faithfully during the last busy weeks of work on the thesis.

Svein Erik Bratsberg taught me much about object-orientation. The guest lecture by Antonio Porto of Universidade Nova de Lisboa at IDT provoked some key ideas in this thesis. Finally, the discussions with Tore Amble provided me with an insight which played the decisive role in completing the thesis. Many thanks for this!

My special thanks must go to Utlendingsdirektoratet (UDI) of Norway whose psychological pressure (i.e. their keen will to expel me from the country) guaranteed the permanent frustration I had to be under, while writing the thesis. It is due to them that the thesis is far not as ripe as it should (and—I dare hope—could) have been, and that it involves no implementation of the formalism proposed in it.

In my job of fighting UDI, I had the luck to experience kind and strong assistance by Per Lund from Radmann Service AS in Trondheim. I also got various helpful support from Arne Asphjell and Anna Komorowska, both of Internasjonal avdeling, NTH, and from my two supervisors.

Last but never least, I cordially thank Marit, Åsmund, Øystein, and Teno for their friendship throughout my days in Trondheim.

Contents

Chapter 1

Introduction

1.1 Motivation, Goals and Results

The purpose of the present thesis has been to study possibilities to combine the object-oriented (OO) and the logic software development paradigms. These two paradigms differ vastly in many respects.

- OO offers modularity, encapsulation and sharing. Logic offers reasoning and deductive retrieval via unification and backtracking. The two paradigms exhibit different advantages and disadvantages, and the hope to benefit from their synergic merits has given an impetus to numerous studies on their merging perspectives.
- But even more significantly, the paradigms differ in their initial motivation. OO strives to ease software writing—to conduce reuse etc.—i.e. to provide *convenience*, and is generally very practically-minded. The logic paradigm puts more stress on *uniformity*, on some inner clarity which may appear hardly noticeable for an outsider, and seems to be less concerned about applicability in wide practice, being more ambitious with regards to gaining success as a research tool. The difference in motivation turns out to become an obstacle to merging. In the research communities that work on the two paradigms, we find a respective "cultural" difference, and this is natural, since a software paradigm is similar to a world-view, assuming a certain way of thinking. We do not make assertions about superiority of any of the two (as none of them is superior), but rather intend to suggest that any merging attempt ought to consider this "deeper" difference, and adopt some position in this connection.

The thesis contains an overview of more than 20 implemented or suggestion-level mergers (the "related-work"-part). Most of the mergers fall under the two following categories:

biased mergers There is a bias towards one of the paradigms. This paradigm is ruling, and has provided the base language, whilst the concepts and techniques of the other have been adjusted to become incorporable.

hybrid mergers Both paradigms have undergone major transformations before merged.

Hybrid mergers teach us that flexibility in treating the paradigms often occurs fruitful.

The thesis also contains a proposal towards a new solution. This proposal emerged from a conviction that the convenience and uniformity ideals are badly compatible in principle. To overcome the difficulty, a two-layer architecture of a merger system is suggested, where the internal layer pretends to adhere to the uniformity ideal, while the user-visible front-end layer cares more about convenience. The internal layer is elaborated in detail in the thesis, whereas the design of the front-end layer and implementation have been left for future work.

The internal layer rests on modal logic. At the present stage of development, it allows to reason about static object hierarchies where the objects can be either static or dynamic. All the designed modal logics use the concept of *unit*. Units are chunks of knowledge, and they roughly correspond to possible worlds in Kripke semantics.

For the case of static objects, a logic $\mathrm{MU}_{\mathcal{U}}$ (**M**odal **U**nits), and its advanced variants $\mathrm{MU}'_{\mathcal{U}}$ and $\mathrm{MU}''_{\mathcal{U}}$, allowing different varieties of inheritance modes, have been designed and studied. In these logics, units embody objects. Dynamics has been treated in a logic $2\mathrm{MU}_{\mathcal{U}}$, and in its corresponding advanced variants. Their design proceeded from a belief that in OO essentially two orthogonal but similar dimensions of evolution are involved—object hierarhy and time. In these logics, units correspond to pairs $\langle$object, time instant$\rangle$. According to the mentioned belief, the essential distinguishing characteristic of methods and attributes is that the former are inherited along the object hierarchy, whereas the latter are inherited along the (usually linear) hierarchy of time instants.

For all the named logics, if not more then at least the resolution calculi are given. Their pragmatics is discussed along with examples.

1.2 Organization of the Thesis

Chapter 2 serves as an outline of the OO paradigm and as a platform for further presentation.

Chapters 3 and 4 constitute the main part of the thesis. In Chapter 3, an overview and a comparison of a collection of existing proposals and solutions in combining the two paradigms are given. Chapter 4 contains a description of the author's own proposal. Most attention there is paid to elaboration of the logics of the internal layer of the two-layer merger architecture.

Chapter 5 draws some general conclusions and points out some directions for future work.

1.3 Assumption about the Reader

It is assumed that the reader has a computer science background. Knowledge of the basics of both of the two paradigms under study will be necessary. Though Chapter 2 is an outline of the OO paradigm, it is meant more to fix a proceeding point for further presentation than to provide a comprehensive introduction to the field. The thesis contains no introduction into the logic paradigm.

The internal layer of the two-layer architecture proposal that we make in this thesis, rests heavily on modal logic. Section 4.5 provides the most necessary preliminaries, but concentrates only on the distinctive features of modal logic from classical logic. Some acquaintance with (general) proof theory and model theory would therefore be of help.

1.4 Terminology

The OO terminology that we use throughout the thesis, will be fixed in Chapter 2. There is much confusion in the basic terminology of this field currently, and we will simply try to keep hold to one of the possible sets of terms.

The terms of logic and logic programming that we use, are standard. It must be noted only that preference is given to logic terms, when speaking about "pure" logic, and to logic programming terms, when speaking about "applied" logic. In connection with axiomatic calculi, we thus use the terms of logic, whilst in connection with resolution calculi, we choose to adhere to logic programming terminology. The following two pairs of terms require special attention:

Logic	Logic programming
individual constant	atom
atomic formula	literal

Also, the implication "if A then B" is written either as $A \to B$ or as $B \leftarrow A$, dependingly on whether the formula occurs in the context of "pure" or "applied logic". Letters p, q, r usually stand for atomic formulae, whereas A, B, C denote arbitrary formulae.

As regards to modal logic, the terms and notation of the textbook by B. F. Chellas [Che80] are followed mostly.

Proof trees are written with the root (the theorem) in the bottom.

Chapter 2

Object-Oriented Paradigm

During the recent years, OO has become increasingly popular. As B. Stroustrup, the designer of the C++ language, has expressed, " "object-oriented" has become a hich-tech synonym for "good" "[Str88]. One of the circumstances that enables people to label many products as OO, is a lack for a universally accepted definition of OO. There are several reasons behind this confusion:

- The OO area is relatively new, and hence in a status of ongoing developments. Though Simula67 stems from the mid-60s, explosive interest in OO started during the 80s.
- Highly abstract concepts (and paradigm-scale concepts definitely are abstract) tend to lack clear boundaries. Generalizations can last forever. One faces a dilemma: either to proclaim a notion completed, negating further improvements, and thus reach a fruitless standstill, or to allow its further expansion, being aware of the danger that some day it may extensionally incorporate almost everything, and intensionally make no sense. The more we win in generality of a concept, the more we loose in interesting properties common to all its exemplars. To find the reasonable borderline is not so easy.
- Much of the fuzziness comes from disagreement about lower-level concepts. People misinterpret each other. There is less philosophical excuse for that, but it still will take some years till the basic terminology reaches stability.
- Some flexibility is never too bad: it helps to avoid falling into dogmas. This especially important with fastly evolving areas.

Several attempts have been made to unify the views on OO, e.g. in [Str88, KM90, BGHS91]. In the following sections, we enumerate the most characteristic features of a software development paradigm and of the OO paradigm in particular.

2.1 What Is a Software Development Paradigm?

In software development, the word '*paradigm*' usually indicates how a software engineer perceives the "art" of producing software, emphasizing the "images" he considers to be primary about programs. To some, programming is basically about procedures and control flow. This may bring them into the *procedural paradigm*. To others, computation of the desired value via nested functions is the most important (the emphasis is more on data flow). That might lead to the *functional paradigm.* Even others concentrate on relations between data items, thus appreciating the *logic paradigm.* And fourth could view program execution as communication between relatively autonomous entities named "objects", exploiting the *OO paradigm.* An elementary comparison of different paradigms has been presented in [BS86].

Note that a paradigm <u>need not be</u> (and hardly ever is) universal to fit all sorts of application areas. Each paradigm specifically suits for some areas, but may be completely inadequate with respect to others. The logic paradigm, for instance, supports tasks involving deductive retrieval but disencourages numeric computations.

There are at least five generally accepted *phases* of software lifecycle: *requirements specification, analysis, design, implementation*, and *maintenance.* Again, different paradigms are not equally suitable for all of them. The idea of logic might support requirements specification but prove ineffecient later. Functional programming fits many kinds of problems but produces poorly readable programs (deep nestings!), being difficult to maintain. The OO paradigm claims to be universal in this respect [KM90, HE90]. We return to this point later.

Different paradigms need different *languages* and *tools*—compilers, software engineering environments—to support software development "in accordance with them". These must encourage, not merely enable to work within both the *facilities* and *discipline* (purposely introduced limitations) foreseen by the paradigm [Str88].

A software development paradigm can be compared to a world-view. It is relatively easy to work fluently with a couple of languages, but to reorient oneself onto a new paradigm is hard. And even if one succeeds, quick switches back and forth remain still difficult. This demonstrates that any merging of paradigms requires two aspects to be handled. First, theoretic possibilities to perform the merging, and second, practical useability of the proposals drawn from theoretic considerations. Besides, last not least, there has to be a general idea about the overall purpose of the merger.

2.2 What Is Object-Orientation?

Many authors keep satisfied with a claim that object-orientation is basically just thinking in terms of *objects, classes* and *(class) inheritance*, and do not seek for any deeper principles behind OO. This also is finds reflection in a rather widely accepted sliding scale towards object-orientation:

object-based: objects (communicating, hopefully),

class-based: object-based plus classes,

object-oriented: class-based plus inheritance.

In contrast to that, a broader view is considered in [BGHS91]. Chapters 1,2,4,5 of that book present a uniform and rather philosophical basis which is further supplemented by detailed studies on languages, databases, distributed systems and user interfaces, and also on the applicability of OO to the design phase of software lifecycle. Unfortunately, the carefully and systematically introduced terminology of the first chapters is not always adhered to in these further studies, e.g. with regards to polymorphism, and distinctive features of set, class and type.

In [KM90], a different unifying OO paradigm is suggested, where the principles and basic concepts of OO, and the problems of lifecycle model and tools are considered in parallel.

2.2.1 General Principles and Framework

In [BGHS91] four principles are suggested as measures of the “rate of object-orientedness”. These are:

- data abstraction,
- behaviour sharing,
- evolution,
- correctness.

Data abstraction encompasses two aspects: modularization and information hiding. *Modularization* is concerned with stepwise breakdown of complex systems into self-contained and manageable entities, and it is crucial for coping with complex systems. *Information hiding* is a further stage of abstraction, hiding implementation details from the client and restricting access to each module by a protected and declared interface. The interface of a module (the discipline enforced) is called its *behavior*. Sometimes, behavior is also referred to as *specification* to emphasize its difference from *implementation* (the hidden part of the module).

Another pair of principles is used [KM90] to express approximately the same ideas. *Strong cohesion* and *weak coupling* are guidelines for modularization. A module should be sufficiently self-contained and independent on other modules. Coupling of modules is further weakened through information hiding.

Behavior sharing deals with interfaces. It might be desirable to have modules with similar interfaces. To provide this, (flat) *classifications* and (hierachical) *taxonomies* are utilized.

Evolution in OO involves two levels. *Requirements evolution* means stepwise (top-down) refinement of the specification. *Solution by evolution* is the inverse: a synthesizing activity, building up a solution (bottom-up) in an incremental manner. The OO philosophy is to provide a uniform approach for the whole software lifecycle [HE90]. In [BGHS91], it is even claimed that the evolutionary approach is unique to OO computing. Though the latter statement is disputable—stepwise prototyping, for instance, is thinkable also within other paradigms—the evolutionary feature of OO still provides powerful support to exploratory programming.

Correctness is the most “questionable” principle of OO, since actually it is hard in OO to get convinced about correctness. As many decisions are made dynamically (at run time) in OO, it is not always possible to predict the execution details at compile time. This lays certain requirements on *exception handling*. Another issue is concerned with semantic correctness (w.r.t. requirements specification), i.e. *verification*. Still, it is not clear whether correctness really should be listed among the basic principles of OO, since though an important problem, it is not unique to OO. Correctness has been under research in more traditional fields—like concurrent programming or embedded systems—and treated much more seriously there.

Correctness is in no way opposite to flexibility. Though evolution allows different levels of “precision” in different stages, it is possible (and even desirable) to maintain correctness with respect to a given level at each of these stages.

Besides the above principles, the so-called “framework” is formulated in [BGHS91] to systematize the techniques of OO. *Encapsulation* is a generic term for techniques realizing data abstraction, so it is in an one-to-one correspondence to the data abstraction principle.

Techniques that realize behavior sharing, fall under two categories—*classification sharing*, which is more traditional, and *flexible sharing*, which includes more powerful and dynamic techniques (and therefore also realizes the evolutionary principle of OO). Lastly, there are *interpretative techniques* which resolve the previous ones on the implementational level. They tend to vary according to the timing of interpretation, i.e. at compile or run time. However, the framework classification of techniques are of auxiliary value, helping to see which principle a particular concept or technique is utilized to realize.

2.2.2 Basic Concepts and Techniques

In this subsection, we briefly summarize the essence of the basic concepts and techniques of OO, mainly basing on the approach of [BGHS91].

Objects

Objects are the fundamental building blocks in OO. At the conceptual level, an object is any perceived entity of the system being developed. In the first approximation, objects represent real world entities. Actually, objects may also represent abstractions without direct counterparts in the real world.

In terms of physical realization, objects map directly onto the framework category of *encapsulation*—an object is an encapsulation of a set of *operations* or *methods*, which can be invoked externally, and of a *state* (stored in *(state) attributes*), which remembers the effect of the methods. The methods and variables of a given object together constitute its *properties*. Synonymously to '(state) attribute', also the expressions '(state) slot' and '(instance) variable' are frequently used.

Attributes are private to the object they characterize, and inaccessible directly for other objects. They are the ultimate realization of encapsulation. Methods serve as the interaction interface of the object. Thus, object interaction is equivalent to remote invocation of methods. In OO, invocation is often described as being achieved through *message passing*—requests by an object for a method of another object (or self) to be invoked are communicated in a form of messages. Physically, no messages need be sent in a system with message passing. Equally, some controlled form of procedure calls is possible. Though the syntax for message sending differs from system to system, the following components are generally present: (*sender-object*), *receiver-object*, *method-name*, *parameters*.

Classes

Classes support the *classification sharing* category of the framework. They allow groups of objects to share properties. A class serves as a *template* from which objects may be created. Objects belonging to a class are called its *instances*. In a class, the descriptions of the attributes (state descriptors) and the definitions of the methods of its instances are stored. The values of attributes are stored in instances themselves. Note that though all the instances of one class exhibit common behavior, they are not identical since the values of their attributes can be different.

In class-based systems, given a template (in a form of a class definition), objects can be created (*instantiated*) dynamically. Instantiation is carried out by asking a class to create a new object according to the template. The object is then initialized with a state (attribute values) determined by the request parameters.

Inheritance

With classes alone, all class definitions in the library of classes would have to be written out fully. In many cases, this would be an unnecessary waste, because similar classes exist in a system. *Inheritance* means a part of the behavior of one class being defined in another, and it is a way of structuring and "slimming" the library. The inheriting class is said to be a *subclass*, and the inherited class is said to be a *superclass*. In *strict inheritance*, a subclass inherits all the attribute descriptions and method definitions stored in a superclass (plus may have some of its own). In *non-strict inheritance*, a part of the behavior of a superclass can be changed (overloaded) or deleted in a subclass.

As an OO system develops, subclasses are constructed out of existing classes until the appropriate functionality is developed. As a result, a *class hierarchy* emerges. Normally, the hierarchy is rooted by a special built-in class (usually called `object`) which contains attribute descriptions and method definitions common to all classes.

Subclass/superclass relations are often described as *isa* links. Correspondingly, the term *isa hierarchy* is used synonymously to class hierachy.

One of the consequences of inheritance is that the complete knowledge of an object's properties is no longer concentrated into a single place (its class)—in order to execute a method, its appropriate definition must be searched in the class hierarchy. In the present thesis, the class who possesses this definition, will be often called its owner (following the terminology of [DH91]), in contrast to its inheriting classes. Association of a method name to a definition (implementation) is called *method binding*. Method binding can be performed either *statically*, during compile time, or *dynamically*, at run-time. Static binding is often too rigid, whereas dynamic one causes much computational overhead, and makes code hard to read and verify. Usually the binding is carried out dynamically, thus allowing bindings to change from invocation to invocation.

In many OO systems, a class can have multiple superclasses. The respective inheritance is called *multiple inheritance*, in contrast to the simple *single inheritance*. Multiple inheritance has more expressive power than single inheritance, but causes binding conflicts for properties theat could be inherited from several branches of class hierachy at a time. To solve these, various strategies are used (bottom-up left-to-right in Flavors, for example).

Sometimes special abstract classes, called *mixins*, that cannot have any instances, and act only as superclasses of other classes, are used to enrich the single inheritance mechanism. Mixins exist to add functionality to ordinary classes. In case of binding conflicts, preference is given to an ordinary superclass, rather than to a mixin. In some systems, mixins are used at the level of instances (new functionality is added not to templates, but to instances directly).

Metaclasses

Like similar objects are grouped into classes, similar classes also could be grouped. Several systems support *metaclasses*, which serve as templates for classes. In these systems, classes are objects in their own right. There is a close analogy between the pairs "instance—class" and "class—metaclass". Besides holding descriptions of instance attributes, classes can have their own, *class attributes*. Their values of are stored in their possessing classes, but descriptions are stored in their metaclasses. Classes can also have *class methods* with definitions stored in their metaclasses. The most typical of such is `new` which, when being invoked on a class, causes instantiation of a new object.

Class attributes are seen by every instance of the class: their values are available to (instance) methods held in that class. Like instance attributes, they are inherited along the class hierachy and visible to subclasses. Unlike instance attributes, also their values (not just descriptions) are visible to subclasses.

Each metaclass is a class in its own right, and thus must possess a metaclass. In this way, besides the class hierarchy, a *metaclass hierachy* arises. Class methods are inherited along the metaclass hierarchy. The metaclass hierarchy can be rooted by a special class (most frequently called `class`) containing definitions of class methods common to all classes (e.g. `new`).

Metaclasses add much power to an OO system, but they also may make it complicated. Their main mission is to provide uniform treatment of objects and classes, and to create system-level self-containedness. A classic language supporting metaclasses is Smalltalk80.

Delegation

In so-called *classless* OO systems, no distinction is made between classes and objects; only objects are dealt with. In these, *delegation* may be utilized to accomplish property sharing and binding. Delegation allows incremental definition of all objects, enabling attribute values and method definitions to be shared. If an object delegates some properties—attributes or method—to another object (its *prototype*), then they will be stored only in the prototype. Any changes to those properties will affect both the object and the prototype. In this way, the object and the prototype mutually depend on each other.

In [Ste87], it has been proven formally that inheritance and delegation can model each other to a certain extent. The proof that inheritance can model delegation, is based on the observation that strict inheritance is “delegation on the class-level”. The objects of a classless system have to be mapped into classes without objects, attributes into class attributes, methods into (instance) methods (defined in classes), and the object/prototype relation into the subclass/superclass relation.

Delegation cannot model inheritance so precisely, because in delegation, attributes cannot be described in one object, and have their values stored in another, which is exactly what happens to instance attributes in class-based systems (they are described in classes, but have their values stored in objects). Still, a concept of two models being equivalent up to instance templates can be defined, and inheritance can be modelled by means of delegation to the precision of this equivalency.

Types. Abstract data types

A *type* (in general, not merely in the OO context) is an abstract description of a related group of entities. In an untyped world, it would be impossible to reason about individual entities, as they would all appear different and unconnected. Typing enables similar entities to be grouped, so that similarities can be promoted and differences ignored. Three different roles of typing can be identified:

1. types are abstracting over the underlying properties of entities,
2. they allow higher level abstractions (e.g. more complex types) to be developed,
3. and they provide a level of protection against incorrect or undesired actions.

The latter role of type system is called *type checking.* The two possible sources for type errors are parameter passing and assignment. Type checking, just as method binding, can be *static* or *dynamic*, according to its timing.

Types in OO, like elsewhere, abstract over the underlying properties of entities, i.e. over objects, in the case. All private properties of an object are encapsulated within the object, and this encapsulation is protected behind an abstract interface which we term as behavior. An OO type, consequently, is a collection of objects with the same behavior, i.e. with the same set of methods (more exactly, method specifications, not their implementations).

Subtyping is concerned with behavior sharing. A type T is a *subtype* of a type T' iff T provides at least the behavior of T'. An object of the type T can then be used as if it were of the type T'. Note that type is not class, nor is subtyping the same as subclassing. Two objects of same class must have identical method implementations. To belong to same type, only specifications of their methods must coincide.

Abstract data types (ADT's) are an advanced development of the principle of data abstraction. This principle emerged in the mid-1970's as a major technique in handling complexity. However, the roots of ADT's are older, curiously, and can be traced back to Simula67. ADT's extend the principle of data abstraction by separating the specifications of data abstractions from their implementation. ADT's thus can be considered to consist of two parts: the *specification part* and the *implementation part.* The former incorporates the syntax (the signature) and the semantics (operational or denotational (algebraic)). The latter is given by a representation (data structures) and associated algorithms. Subtyping in ADT's is defined through the *conformance rules.*

Polymorphism

Polymorphism provides flexible typing disciplines. It is defined as an ability for an entity to act as of more than one type. According to a classic taxonomy by Cardelli and Wegner [Weg87], techniques of polymorphism can be divided into ad hoc and universal.

Ad hoc techniques are older and work only on a specific number of types in an unprincipled way. The best known of these techniques are coercion and overloading. *Coercion* means having built-in mappings between some types (like from `integer` into `real`). *Overloading* allows a function to have different definitions for different types of parameters (like `add` being defined for both integers and reals).

The two techniques of *universal* polymorphism have more relevance to OO. In *parametric polymorphism*, a function (coded once) can work uniformly on a range of types. Type is left open (i.e. it is a parameter) in the function definition code, and is instantiated at each call in accordance to the type of the actual argument. Parametric functions are sometimes called *generic* functions. *Inclusion polymorphism* is more limited: it allows a function to operate on a range of types determined by subtyping relationships.

Inheritance involves two kinds of polymorphism. First, a method defined in a particular class is automatically defined for all its subclasses, and identically thereby, in the general case (*subclassing*). Second, it is also possible to *override* this sharing through redefining the method in a subclass, thus *overloading* its implementation from the superclass. Both kinds can be considered to fall under inclusion polymorphism. Besides that, the second kind is clearly a form of the ad hoc overloading, too. On one hand, the label 'ad hoc' is justified in the case, since, indeed, universal redefinition generators do not exist. On the other hand, a doze of universality is also present, because the binding mechanism itself is universal in inheritance (the inherited definition is the default, and the new definition is the explicit).

2.2.3 OO and Software Lifecycle

The interest towards OO in the software engineers' community has been gradually increasing. OO is often claimed to be equally suitable for all the phases of the software lifecycle. The evolutionary principle of OO is in correspondence with the very idea of systematic development. The arguments on the benefits of this circumstance from [HE90, KM90], plus some of ours, can be briefly summarized as follows:

- As OO applies to all lifecycle phases, no "translation" from the output documentation of one phase into an input document of the next phase is needed. Moreover, such output can possibly even serve as an initial layer for the next phase.
- The above-mentioned diffusion of phases makes their boundaries blurring.
- The lack of distinct boundaries and the *iterative nature* of OO development (the *"fountain model"* of software lifecycle [HE90]) amplify each other. One one hand, due to the diffusion, reiteration in the software development process is not as painful as it is in more traditional paradigms. On the other hand, the same reiteration even more blurs the boundaries.
- The similarity between *semantic data models* (SDM's, the Entity-Relationship Model being the simplest) and OO is of their mutual benefit. First, it has given an impetus to OO databases. Second, it allows one to think in SDM categories when doing OO analysis and design.
- Encapsulation, classing and flexible sharing all encourage *software reuse*. Thus, the fate of always starting up from scratch can be escaped.

Chapter 3

Reported Mergers

In this chapter, our attention is paid to the existing proposals and solutions in combining the two paradigms. Section 3.1 discusses the objectives for combination. Sections 3.2-3.5 give an overview of a collection of mergers. In Section 3.6, these mergers are compared and categorized.

The set of mergers, considered below, does not pretend to be complete, but rather to illustrate the variety of the approaches taken.

3.1 Merging Aims

The main objective to merge two paradigms is to exploit the synergic merits of both, and to eliminate the disadvantages of both. Significantly, authors of different solutions point out very overlapping pros and cons of each paradigm, though having set forth completely different purposes in their merging efforts [BMS90, FH86, FY88, IC90, KE88].

Modularity, encapsulation and sharing (both classed sharing, i.e. inheritance, and flexible one, i.e. polymorphism etc.) constitute the advantages of the OO paradigm.

Declarative framework with pattern matching, unification and backtracking are the main beneficial characteristics of the logic approach. Even its context-sensitivity (non-monotonicity, in other words) which usually causes trouble, may appear as a pro, if we speak about production rules. The main disadvantage of the logic approach lies in the flatness of clause base, which makes deduction ineffective and reduces modifiability.

The first area where a demand for joint exploitation of both paradigms appeared, was knowledge bases. On one hand, there was a need to structure knowledge, to build up the architecture hierarchically. On the other hand, effective deductive retrieval via pattern matching was required. Many of the solutions considered below, stem from practical applications in knowledge handling.

Two main approaches to combining the two paradigms can be observed. The first is to modify the basic ideas and concepts of one paradigm so that they become incorporable into the other in a more or less natural way (*biased mergers*). The second is rather to bring them together into a synergistic unification under a new subsuming paradigm, which may require both original paradigms to undergo major transformation (*hybrid mergers*). The second approach has been underlined in [BMS90], though it is disputable whether it actually is followed there.

In [GM87], concern is expressed about the dangers of integration for the sake of integration, stating that “experience (with languages like PL/I) has shown that patching together features from different paradigms in an ad hoc way results in a complex language without intellectual coherence”.

Most of the solutions reported in literature, are biased. One paradigm has remained ruling and provided the basis language, while features from the other one have been adjusted to suit that context. In one case, a “language interface” was reported. Hybrid mergers are of special interest.

3.2 Introducing Logical Features Into OO Languages

3.2.1 Orient84/K

The language Orient84/K [TI84, IT86], designed at Keyo University, Japan, relies on the so-called distributed knowledge object modelling (the authors' method for OO knowledge systems). The syntax and semantics of Orient84/K owe much to, and are extended from Smalltalk80. It has the metaclass/class/instance distinction, and allows multiple inheritance. In order to provide a combination of OO, demon-oriented (which really means access-oriented), concurrent programming and logic paradigms, each object consists of monitor, behavior, and knowledge-base parts.

A class definition has the following form:

`CLASS` *class-name*
`INHERITS FROM` *superclasses*
`CLASS SECTION` *section-body*
`INSTANCE SECTION` *section-body*

where *section-body* is of the form

`OWN VARIABLES` *own-variable-definitions*
`MONITOR PART` *monitor*
`BEHAVIOR PART` *method-definitions*
`KNOWLEDGE-BASE PART` *Horn-clauses*

The term ‘own variable’ corresponds to ‘attribute’ in our terminology.

The functions of the monitor part are access control, prioritized message handling, and statistics gathering. Priotized methods allow to encompass access-orientation—active values are specified in the monitor part.

The behavior part contains method definitions. In order to enable to add and delete methods to and from objects dynamically, the methods `add_method` and `delete_method` have been predefined.

The knowledge-base part determines the local knowledge base of an object. It can be viewed as a collection of the object's special own variables (i.e. instance attributes) storing rules and facts, since it can be changed through message passing. On the other hand, the visibility of rules and facts along the inheritance path is equivalent to that of methods. In the knowledge-base part, three kinds of variables can be used: own, K-formal, and temporary variables. Both the K-formal and temporary variables correspond to logical variables in logic programming. The difference is that K-formal variables are used to pass information between the behavior part and the knowledge-base part, whereas temporary variables are local to the knowledge-base part. K-formal variables are prefixed by `?`.

Interaction between the behavior and knowledge-base parts is accomplished via the predefined `unify` and `foreach_unify` methods.

In the following example, a method which sends mail to all brothers of a person, is defined:

```
OWN VARIABLES
    john tom mike andy peter henry robert
BEHAVIOR PART
    sendToBrothersOf: m mail: mess1 |x|
        foreach_unify(brother(m,?x))
            do:[:x| -> x sendMail: mess1].
KNOWLEDGE-BASE PART
    "RULE"
        brother(?x,?y) |f|
            father(?x,f),father(?y,f).
    "FACT"
        father(john,henry).
        father(tom,robert).
        father(mike,henry).
        father(andy,robert).
        father(peter,robert).
```

The following example demonstrates how a knowledge base can be changed. In order to acquire facts about the mother-child relation, a method for querying mothers about their children is defined.

```
askMother |m x|
    foreach_unify(name(?x))
        do:[:x| m <- x mother.
            appendKB(mother(x,m))].
```

3.2.2 CBL = Constraint-Based Language

The language CBL [BMS90] has been designed at Columbia University, NY, USA. It is based on CLOS (Common Lisp Object Specification System). It pretends to be the first instance of a new, *constraint-based* paradigm (different from what is called constraint-based in [BS86]!), a synthesis of the OO and rule-based paradigms. Though the approach can be worth the label, CBL strongly relies on CLOS. Constraints in CBL are formulated through predicates. The three underlying ideas of the constraint-based paradigm are:

- *Constraint-based invocation*: method invocation is dispatched ("guarded") by programmer-defined constraints.
- *Instance inheritance*: understood as an analogous notion to class inheritance in the sense that if class inheritance structures classes (builds up a hierachy of them) then instance inheritance does the same with instances. Instance inheritance in CBL is grouping of objects into collections defined by constraints, rather than being delegation.
- *Procedural attachment*: functions can be bound (attached) to attributes, objects, classes and collections that will always be invoked when these are accessed (i.e. addressed to or modified).

An example of CBL is the definition of a method computing the greatest common divisor of two numbers:

```
(defmethod gcd (n m) (:constraint n {> m})
    (gcd m (mod n m)))
(defmethod gcd (n m) (:constraint n {= 0}) m)}
```

The authors of CBL point out a similarity between constraints and Dijkstra's *guarded commands*. There is also a relation to *constraint logic programming* (for a survey on the perspectives of the latter, see [JL88]). A careful study on this connection might be worthwhile.

3.2.3 KSL/Logic

The language KSL/Logic [IC90] is a product from Detroit R&D, MI, USA, and is an extension to KSL. The latter is a "pure" object-oriented language where "everything is object" (variables, methods, main programs etc. incl.). All objects in KSL belong to classes, and all activities are initiated by message-passing. To construct KSL/Logic from KSL, a subclass `PredicateBehaviour` has been added to the class `Method` and a class `BuiltInPredicateBehaviour` has been added to the class `ClassBehaviour` in order to facilitate logic programming. Additions to the class `ExpressionObject` also have been made.

The resulting language allows to use logical expressions like logical (i.e. unifyable individual) variables, predicates, Horn clauses, cut etc. KSL method calls can appear as members of Horn clauses. This is a helpful feature in incorporating procedurality into the deductive part of the language. Two restrictions apply in this connection: first, the logical variables in a KSL method expression must become instantiated before the actual invocation of the latter; and second, the side effects of KSL methods will not be removed when deduction comes up to backtracking.

The `ForAll` expression is an inverse link between the procedural KSL code and logic, and allows passing deduction results (answer substitutions) to KSL procedures.

The following examples illustrate the use of the above features:

```
(HC
    (P getLoan %Person %Amount)
    (P bankLoan %Person (Debts %Person) %Limit)
    (LessThan %Amount %Limit)
    (Print "Loan Available"))
```

`HC` stands for Horn clause, `P` for predicate, and `%` is used to denote logical variables. `Debts` refers to a programmer-defined KSL method.

```
(ForAll $Self (P newmail %QualifiedPerson %Age %Income)
    (#List
         (Print (Name %QualifiedPerson))
         (Print (Address %QualifiedPerson))))
```

A successful resolution of the `newmail` predicate behavior will bind a qualified person to the `%QualifiedPerson` variable, basing on a given age and income. For each of these resolutions, the KSL `Print` message will access the value bound to the `%QualifiedPerson`.

3.3 Interfacing Languages of Different Paradigms

3.3.1 Prolog/Loops

The article [KE88] reports on a language interface designed at Xerox AI Systems, IL, USA. It is ambitiously titled "Bridging the gap between object-oriented and logic programming". Unfortunately, the solution is more like "filling in", rather than elegantly bridging the gap. An interface has been established between Loops (the Xerox AI environment of OO programming) and Xerox Quintus Prolog. What has been implemented, is a facility to make calls to Loops objects from Prolog and—vice versa—to set Prolog goals from Loops (treated as messages in the latter). Finally, Prolog clauses can be treated as Loops objects.

To perform a slot-value substitution of a Loops object (i.e. to change the value of one of its attributes) from Prolog, the following Prolog clause might be exploited:

```
substitute_slot_value(Object,Slot,Old_value,New_value):-
    get_value(Object,Slot,Old_value),
    put_value(Object,Slot,New_value).}
```

A Prolog goal

```
?- pred(arg1,X,arg3).
```

could be invoked from Loops calling

```
(PROLOG 'pred (LIST 'arg1 *VALUE* 'arg3))
```

Though a language interface like Prolog/Loops involves no real integration, it advantageously imposes no degradation of performance, since none of the languages has been implemented on top of the other. Generally, disturbing overhead is often the case when integration is achieved through introduction of the features of one paradigm into a language of another.

3.4 Introducing OO Features Into Logic Languages

3.4.1 Extension to Prolog by Zaniolo

The extension to Prolog suggested by C. Zaniolo, Bell Laboratories, NJ, USA [Zan84], is an early attempt to introduce OO features into a logic language. Three `xfx` operators `isa`, `with` and `:` have been implemented on top of UNSW Prolog.

An object definition has the form

object `with` *method-list*,

where *object* is a Prolog predicate with zero ore more arguments, and each method in *method-list* is a Prolog clause. This usage of the word 'object' is not adhering to correct terminology. In fact, the arguments of the predicate associated with *object* correspond to attributes. Hence, the above definition actually "creates" objects only when the argument terms of this predicate become ground, and generally defines classes (templates) only.

The fact

```
reg_poly(N,L) with
    [perimeter(P) :- P is N*L,
        what_is_it(a_reg_polygon)].
```

defines `reg_poly`'s to be objects with number and length attributes, and with `perimeter` and `what_is_it` as methods.

The : predicate is used to imitate message passing in the form form

receiver:*goal*.

For example,

```
reg_poly(6,10):perimeter(X) ?
```

The inheritance network is declared through a predicate `isa` in the form

sub-object `isa` *object*,

where—again—actually subclasses and classes are meant. E.g.

```
rectangle(Base,Height) isa parallelogram(Base,Height).
square(L) isa rectangle(L,L).
square(L) isa reg_poly(4,L).
```

The "states" of objects in this implementation cannot be changed. They are static, and serve also as identifiers to objects—an object is essentially referred to by indicating its class and the values of its attributes. A method declared in a subclass overrides the method with the same name of the superclass. Multiple inheritance is allowed.

3.4.2 Prolog/KR

Prolog/KR [Nak84] is an extension to Prolog, but was implemented originally in versions of Lisp. Its design aim was to provide Prolog with modularization in order to cope with knowledge representation tasks. Prolog/KR differs from other solutions considered in this paper, as it follows OO ideas only vaguely. There are no objects, classes, nor message-passing in Prolog/KR. Instead, the concept of *multiple worlds* is used to structure the clause base. Neither is inheritance hierarchy a primary notion in Prolog/KR. Worlds are linked dynamically in general, though fixed hierarchies are possible.

Prolog/KR adopts Lisp's syntax of S-expressions. All sentences of a program are executed, and they are called *predicate calls*. Clauses can be entered into knowledge base only using `assert`-like predicates. With this syntax, Prolog/KR is able to provide higher-order control predicates like complex conditions, conditional branches, repetitions, accumulation of solutions, process objects, partial concurrency etc.

Prolog/KR worlds are definition spaces. Predicates defined in another world must be called indicating the name of the world:

(`with` *world-name predicate-calls*).

Inheritance of predicate definitions from other worlds is achieved through dynamic nesting of worlds—the innermost definition is used. Suppose that a predicate `P` is defined in two worlds `A` and `B`:

```
(with A (assert (P) ... ))
(with B (assert (P) ... ))
```

Then the meaning of `P` is different in the following two calls:

```
(with A (with B (P))) ; P from B is used
(with B (with A (P))) ; P from A is used.
```

There are two predicates for making assertions in Prolog/KR, and they have different effects when assertions are done in nested worlds. If the `assert` predicate is used, the definition of the head predicate of an asserted clause in a given world becomes the union of all clauses with the same head predicate that are visible from the world at the assertion time. Alternatively, the `define` predicate can be used to override any previous definitions from the outside. By using `define`, a number of clauses can be entered at a time.

The following example clarifies this mechanism:

```
(with A
    (assert (P a1))
    (assert (P a2))
    (assert (Q qa))
    (assert (R ra))
    (with B
        (assert (P pb))
        (define Q ((qb1)) ((qb2)))))
(with A (P *x)) ; *x = pa1, pa2
(with A (Q *x)) ; *x = qa
(with A (R *x)) ; *x = ra
(with B (P *x)) ; *x = pa1, pa2, pb (A+B)
(with B (Q *x)) ; *x = qb1, qb2 (B only)
(with B (R *x)) ; no definition in B
(with A (with B (R *x)) ; *x = ra (A only)
```

If some static hierachy is kept in mind, the ugly nesting of `with` can avoided, if it is described as an *AKO* (isa) hierarchy. The `AKO` predicate can be defined as follows:

```
(assert (AKO *x *y)
    (asserta (WITH-WORLD *x *p)
             (WITH-WORLD *y (with *x *p))))
```

After having executed this and the following “declarations”

```
(AKO penguin bird)
(AKO bird animal)
(AKO animal object)
```

the sentence

```
(WITH-WORLD penguin (NUMBER-OF-WINGS *n))
```

will be expanded at run-time into

```
(with object
    (with animal
        (with bird
            (with penguin (NUMBER-OF-WINGS *n)))))
```

Prolog/KR allows multiple inheritance and conditional AKO (cf. collections in CBL, Subsection 3.2.2). Since there Prolog/KR is classless, its inheritance is similar to delegation. An advantage of Prolog/KR is its reliance on a very simple idea of worlds with a neat semantics, that allows natural expansion towards several more complicated concepts.

3.4.3 Mandala

Mandala [FTK+84] was designed at ICOT Resarch Center, Tokyo. It has been implemented on top of KL1. The latter is a logic programming language with stream-AND-parallelism like Concurrent Prolog, and was developed for the Fifth Generation Computers' project Parallel Inference Machine (PIM).

Mandala distinguishes between unit worlds (classes, in other words) and instances (objects). Between these, four kind of links can hold. The `instance_of` link is the usual object/class relation, and the `is_a` link is the traditional subclass/superclass relation. A `part_of` link is used to define a composite instance having lower-level instances as its parts. This link has a metalogical meaning that some theory refers to other theory as data objects. (In OO, the situation when attributes of objects are regarded as objects on their own right corresponds to the case.) The `manager_of` link, similarly to `instance_of`, connects instances and world units, but its meaning is completely different. One `manager_of` link is attached to each unit world, connecting it to its manager. The functions of this manager instance include modifying the axioms stored in the unit world and creating and destruction of instances of the unit world (cf. the notion of metaclass).

The classic predicate `demo` of metaprogramming in Prolog [BK82] has been generalized into `simulate` for the purpose of Mandala, being its concurrent version.

Message passing in Mandala is expressed through *streams*. A stream is a list of messages that the object receives during its lifetime (in actual occurrences of streams in programs, of course, tails of them—yet unspecified future messages—are logical variables). Assertions about a unit-world are stated as facts

unit-world-name(*axiom*).

An instance is implemented by a perpetual process which takes local states as arguments. In other words, an instance is realized as a chain of goal reductions. A goal always takes the form:

`instance(`*instance-name,stream,world*`)`

where *world* conceptually represents a set of axioms, contained in the unit world of the instance. The second and third arguments are always read-only. The internal implementation of `instance` (written in Concurrent Prolog here) clarifies the usage of streams in Mandala.

```
instance(Name,[Message|Input],World) :-
    simulate(Name,Message,World,NewWorld),
    instance(Name,Input?,NewWorld?).
instance(Name,[],World).
```

(The `?` postfix in Concurrent Prolog is read-only annotation.)

Mandala also provides a kind of rule-orientation. Weights can be associated with rules, and certainty factors of deduction results can be estimated on their basis.

3.4.4 ESP

ESP [Chi84] was developed as the system description language of SIMPOS, which is the programming and operating system of ψ (the Fifth Generation Computers' project Personal Sequential Inference Machine). ESP has been implemented on top of KL0, a Prolog-like high-level ψ machine language.

From a logic programming viewpoint, an object of ESP is a set of clauses. This clause set is basically (extensions follow below) the union of clauses defined on the inheritance path of the object. A class may have more than one superclasses. Besides logic language features, KL0 also has Lisp-like constructs corresponding to `cons, rplaca` etc. Based on those, time-dependent state slots have been introduced into ESP, to model the OO concept of attribute.

There are three type of predicates in ESP: local predicates (which guarantee information hiding), built-in predicates, and methods. Methods are distinguished from other predicates via prefixing them by a colon. Inheritance operates on both methods and and slots. Method definitions cumulate—they are disjungated if there are several definitions for a given method along the inheritance path. If some slot name has been declared more than once along the inheritance path for a given object, the object still has exactly one slot with that name.

Traditional non-monotonicity of method inheritance, i.e. method overriding, seems to conflict with the basically cumulative nature of the ESP method inheritance, but it can be introduced via two mechanisms.

First, *cut* can be used to accomplish overriding, as in the following simple example:

```
class bird has
    nature animal;
    instance
        :fly(Bird);
        ...
end.

class penguin has
    nature bird;
    instance
        :fly(Penguin):-!,fail;
        ...
end.
```

(`nature` determines the superclasses of a class. The keyword `instance` has nothing to do with any object instance, rather it serves as a delimiter starting the template part in a class definition.)

Second, the unique ESP *demons'* feature could be utilized. Clause heads in definitions of a method can be prefixed by `before` and `after`. These clauses are said to define *before* and *after demon predicates.* Clauses without such a prefix are said to define a *principal* *predicate*. The *method* itself is physically implemented through a *method predicate*. The body of the single clause of its definition is the combination of following:

1. The bodies of all the before demon predicate definitions conjugated (ANDed), in the order of inheritance.
2. The bodies of all the principal predicate definitions disjungated (ORed), in the order of inheritance (the basic case).
3. The bodies of all the after demon predicate definitions conjugated (ORed), in the reverse order of inheritance.

The usage of demons is illustrated in the following example:

```
class with_a_lock has
    instance
        component lock is lock;
        before :open(Obj) :- :unlocked(Obj!lock);
        ...
end.

class door has
    instance
        component state := closed;
        :open(Door) :- Door!state := open;
        ...
end.

class door_with_a_lock has
    nature door, with_a_lock;
end.
```

(`component` is used to define slots; `is` determines the class of the slot; `!` enables to make remote calls to slots; `:=` means assignment to a slot.)

The example states that if something has a lock, then, in order to get opened, it first has to be unlocked, and only thereafter other necessary "operations of opening" can be performed.

ESP also exploits metaprogramming, but only for defining macros (to simplify writing of arithmetical computations etc.).

3.4.5 LOOKS = Logic-Oriented Organized Knowledge System

LOOKS [MOK84], like Mandala and ESP (see Subsections 3.4.3 and 3.4.4, resp.), owes its birth to the Fifth Generation Computers' project. It has been implemented in DEC-10 Prolog.

A LOOKS class has five properties: superclasses, metaclasses, instance attributes, class attributes, and methods. A method is predicate, defined by a single Horn clause. Multiple inheritance is allowed. Method sending is expressed in the form

receiver<-*method-name,arguments*.

If a message is sent to a given object itself, `self` is indicated as *receiver*. An instance is created by sending a message to the desired class.

Metaprogramming (through the predicate `world_demo/4`) is used to deepen multi-paradigmality. In a concrete application—the LOOKS/Glaucoma medical expert system—there are three worlds: the object-world, which is formalized as described above, the decision-world, containing heuristic knowledge in a rule-oriented language, and the patient-world, being defined by simply making assertions. `world-demo` is exploited to dispatch interaction of the worlds.

3.4.6 SPOOL

The language SPOOL [FH86] was designed at IBM Japan Science Institute. It has been implemented on top of VM/Programming in Logic (an IBM implementation in Prolog). A class in SPOOL, similarly to LOOKS (see Subsection 3.4.5), has five properties: superclasses, metaclasses, instance attributes, class attributes, and methods. A method is a predicate defined by means of a set of Horn clauses. Multiple inheritance is allowed, and method search is done in a Flavors-like manner.

Message passing in SPOOL means to set up a goal for the receiver object, and its syntax is:

receiver`<<`*message*.

The receiver object and `<<` can be omitted, if the target is self. SPOOL allows either the message or the recipient to be an uninstantiated logical variable. The authors find that through the foreseen opportunity to omit the receiver, a feature is created, which might be called *intensional*, in contrast to the Smalltalk's *extensional* one. Instances are created by sending the `new` message to the desired class.

The authors point out that since Prolog exploits the *closed-world assumption*, there is externally no difference between falsity of a defined predicate and a non-existent predicate. In OO, an error occurs when an object receives an unknown message. Since such handling helps debugging, `<<` is also programmed to cause error in a similar situation. If we want to get the Prolog-sense failure, we can use either `?<, <?` or `?<?`, according to our exact intention.

To facilitate access to instance attributes and class attributes, two built-in methods have been provided. One has the form

attr`::`*value*,

and is used for value retrieval. It succeeds, if the receiver object has an instance attribute named *attr*, whose value is unifyable with `value`. The other has the form

attr`::=`*value*,

and is used for value assignment. The effects of `::` and `::=` are irreversible.

In the following example, `snoopy` is an instance of `employee`, and asks permit for a domestic trip from his manager `charlie`, who is an instance of `manager`:

```
class employee has
    super-class root-class;
```

```
        instance-var his-manager;
        methods
           request(Subject) :-
               his-manager :: M &
               M << give-me-approval(Subject,Answer);
        end.

    class manager has
        super-class employee;
        methods
            give-me-approval(Subject,approved) :-
                unimportant(Subject);
            give-me-approval(Subject,rejected) :-
                unreasonable(Subject);
            give-me-approval(pending);
            unimportant(domestic-trip);
            unreasonable(double-salary);
        end.

    employee << new(snoopy, {his-manager: charlie}).
    manager << new(charlie, {his-manager: X}).
    snoopy << request(domestic-trip).
```

3.4.7 POKRS = Prolog-Based Object-Oriented Knowledge Representation System

POKRS [FY88] was designed at Hefei Polytecnic, P.R.C., to serve as an environment for large knowledge systems. POKRS has been implemented on top of the Prolog-KABA language, and is meant to run on microcomputers. The POKRS knowledge representation model is based on OO of Smalltalk80 and Loops, and draws heavily from LOOKS, Orient84/K, and SPOOL (see Subsections 3.4.5, 3.2.1, and 3.4.6, resp.) multiparadigm languages/systems. For some reason, the authors of POKRS distinguish between method-set and local-KB parts in class/object definitions. Method-sets are meant to incorporate more procedural, and local-KB's to incorporate more declarative knowledge, though both are expressed in Prolog syntax. POKRS offers metaclasses and multiple inheritance. Message sending is realized through the predicate `send`(*receiver,selector,parameters*).

3.4.8 POL

The POL [Gal86] proposal seems to have no implementation, and is mostly theoretic. The main principles,set forth while designing POL, were the following. POL is to be a superset of Prolog. A relational framework has to be used to appropriately refine OO concepts such as inheritance and method evaluation[1]. Finally, POL must allow dynamic creation of and hierarchical links between objects, classes, and methods. Attributes are not discussed, as they are claimed to be orthogonal to handling of inheritance, which constitutes the kernel of the proposal.

Literals to represent method calls (or message sending) are written as *receiver*:*goal*, where the predicate of *goal* must have been defined in an associated method definition. To deal

[1]This is in contrast to functional, but also to "chunk" framework; by the latter we mean the style of many suggestions with long fill-in definitions à la `class ... has supers ..., metas ..., vars ..., methods ...etc end`.

with hierarchies, two predicates, *subclass* `isa` *superclass* and *object* `instance` *class*, are provided.

The author distinguishes sharply between the *call/return* and *success/failure* paradigms of message passing. In the call/return message passing, the definition of a method from a specific class always overrides that from a more general class. In the success/failure message passing, if a method call evaluates to false, according to its definition in a given class, search for a "better result" goes on, according to inheritance hierarchy, generally. The definition of a given method in a more specific class overrides those from more general ones only in the successful case. The success/failure approach, taken by POL, coincides with cumulation in ESP, but non-monotonicity is introduced differently into POL. POL also supports the idea that all existing solutions generally are to be found.

Method definitions are spread over facts of the following forms:

> *class* `with` *clause*,
> *class* `withdefault` *clause*, *class* `withdeterministic` *clause*.

The head of *clause* must always be a method call. The `with` predicate is used for writing down the normal definition cases of methods. The `withdefault` predicate is exploited to specify the *default* response. If no other definition clause of a given method was successful, its argument clause will be tried. In the case of some solutions already found, the clause specified in `withdefault` will not be attempted. The `withdeterministic` predicate is meant to accomplish the inverse: its argument clause is prioritized, and hence, if it is attempted and succeeds, no other definition clauses will be pursued. This could have been implemented also through cut, but the author argues that his approach is cleaner.

The different method defining facilities are illustrated in the following example.

```
student_researcher isa researcher
student_researcher isa student
fundam_researcher isa researcher
researcher isa person
franz instance student_researcher
joe isa fundam-researcher
age(joe,35)
age(franz,no_value)
courseworks(franz,oo)
in_team(franz,best)
in_team(joe,best)
topics(best,databases)
fields(joe,logic)
researcher with X:is_aged(Y) :- age(X,Y),y=/=no_value
person withdefault X:is_aged(Y) :- askuser("age",X,Y)
student with X:topic(Y) :- courseworks(X,Y)
researcher with X:topic(Y) :- X:in_team(Z),topics(Z,Y)
fundam_researcher withdeterministic X:topic(Y) :- fields(X,Y)

?- joe:is_aged(Y)
Y=35
        /* single solution, askuser not tried  */
?- franz:is_aged(Y)
Enter age of franz:
        /* default applied */
```

```
?- joe:topic(Y)
Y=logic
        /* databases suppressed, as a stronger preference
           is given to fundam_studies with fundam_researchers */
?- franz:topic(Y)
Y=oo
Y=databases
        /* two solutions */
```

As in SPOOL (see Subsection 3.4.6), the so-called *anonymous* or *extensional* method calls are allowed in POL. In this case, deduction is done on all objects of relevant classes (i.e. `withdeterministic` applied to some object does not suppress further search for solutions for other objects, and—similarly—`withdefault` can apply to some object, although solutions for some other objects might have been found already).

3.4.9 CPU = Communicating Prolog Units

The keyword of the solution originally proposed in [MN87], and further elaborated in [Mel91], is *metaprogramming* (similarly to Mandala, Subsection 3.4.3). While the main task of metaclasses in traditional object-oriented systems is to serve as templates for classes, the role of metaprogramming in logic programming languages is more general [BK82]. Many information hiding and sharing disciplines can be more easily and flexibly implemented using metaprogramming.

Each Prolog-unit represents a chunk of knowledge about a particular domain. It is conceptually an autonomous world able to solve queries asked of by user. To solve a goal, a unit also can query other units. Communication between units is achieved through the predicate `ask(`*receiver,goal,answer*`)`. To each unit a metaunit is associated, which stores metalevel knowledge, usually concerned with appropriate search disciplines.

A unit works as follows. If it has to demonstrate a goal, it asks its metaunit for `todemo(`*caller,goal*`,Result)`. If `todemo` succeeds and `Result` is `true`, the substitutions made during deduction are communicated to *caller*. If `todemo` succeeds, and `Result` is bound to a value different from `true`, or if it fails, the unit considers *goal* as failed and performs backtracking.

The main task of the metaunit is to decide how to solve *goal* (cf. managers in Mandala). It can do it directly, ask a different unit for the solution, or send a request for *goal* or a different goal back to *caller*. The latter case is expressed by invocation of the `odemo(`*caller,goal'*`,Result)` predicate. The `odemo` predicate always succeeds.

As an example, consider how method search and default overriding can be implemented through metaunits.

```
todemo(Caller,Goal,Result):-odemo(Goal,Result).
todemo(Caller,Goal,Result):-
    my_name(Myself),          /* my_name finds the name of */
    super(Myself,Mysuper),    /* the current receiver unit */
    ask(Mysuper,Goal,Result).

metaunit(penguin).
  todemo(Caller,[fly],false):-!.
    /* any attempt to demonstrate penguins flying has to fail */
  todemo(Caller,G,R):-ask(birds,G,R).
```

```
        /* everything else about penguins holds with birds as well */
```

Metaunits can be used to define mechanisms for event synchronization, asynchronous activation, and can assist in handling other aspects of concurrency.

In a more advanced variant, a unit can have several metaunits. Also, *instances* are introduced to enable dynamic states. Thus, units appear to correspond to classes, and instances act like objects. CPU instances change their state not by using assignment, but by recursively calling themselves with different argument values. To facilitate communication between instances, special *queue units* are used to serve as communication channels. By that, senders and receivers of messages are subordinated to auxiliary *request sender metaunits* and *request server metaunits.*

3.4.10 Vulcan

The language Vulcan [KTMB87] was designed at Xerox PARC, CA, USA. It is a higher level language, translating into Concurrent Prolog. It draws heavily from the approach by Shapiro and Takeuchi [ST83] to OO in Concurrent Prolog, thus basically being a syntactic sugaring to remedy some syntactic awkwardness and verbosity of OO programming in the "bare" Concurrent Prolog. The key ideas behind Vulcan are similar to those of Mandala, but there is no meta-interpretative processing at run-time in Vulcan. Instead, Vulcan is a preprocessor for Concurrent Prolog.

On the Concurrent Prolog level, objects are represented by perpetual processes consuming streams, where streams serve as message communication channels. On the Vulcan level, this basis is invisible, and the awkward and mistake-provoking syntax (in much consisting of various repetitions), characteristic to stream programming, can be escaped from.

The form for a class definition is

`class(`*class*`,[`*inst-var1,...,inst-varN*`],[`*super-class1,...,super-classM*`]).`

Attribute names must begin with capital letters, and they are used as ordinary logical variables, expect for their scope being not limited to just a single clause, but extending to all method definitions and message sends, associated with that class and its objects. The superclasses' list can be omitted, if empty.

For method defining, there are two ways:

class`::`*message-pattern.*

and

class`::`*message-pattern* `-->` *body.*

Messages are terms, and so are message patterns. *body* is a collection of message sends, separated by commas. If patterns from more than one definition unify with the call on a method invocation, one definition is chosen non-deterministically. When a definition with a body is chosen, the statements of the body are executed concurrently. An assignment operator `new` preceding an attribute name, refers to the attribute's value after the update caused by an invocation. The inheritance mode in Vulcan is overriding; multiple inheritance is provided.

Message sending is written as

object:*message1*:...:*messageN*,

and this queues *message1* through *messageN* for *object* in order. A special object variable `Self` can be used in the bodies of method definitions as a reference to the receiver object. Another object variable, `Super`, can be exploited to bind a send to a definition from the superclasses, rather than from the class of the receiver.

Instance creation is accomplished through a special class method `make` in the following manner:

class:`make([`*value1*,...,*valueN*`],`*new-object*`)`,

where *new-object* is a name beginning with a capital letter, acting as a logical variable with wide scope analogously to instance variables. At the creation, the new object is bound to that variable.

Besides that messages can be sent, they can also be *delegated.* Then `Self` remains bound to the sender object. The syntax is

`delegateTo(`*proxy*,*message*,*client*`)`.

As an example of Vulcan and its underlying mechanism, we show the following fragment:

```
class(window,[X,Y,Width,Height,Contents]).

window::moveBy(DeltaX,DeltaY) -->
    new X is X + DeltaX,
    new Y is Y + DeltaY.
```

This translates into Concurrent Prolog as follows:

```
window([moveBy(DeltaX,DeltaY) | NewWindow],
                   X,Y,Width,Height,Contents) :-
    plus(X,DeltaX,Result1),
    plus(Y,DeltaY,Result2),
    window(NewWindow?,Result1?,Result2?,Width,Height,Contents).
```

Vulcan provides several simplifications in writing arithmetic computations.

As an additional feature, messages in Vulcan can be sent not to objects only, but also to terms. Terms appear effective in modelling immutable (static) objects: if attribute variables do not change, there is no need to store the message stream within an object to distinguish between its revisions.

3.4.11 SCOOP = Structured Concurrent Object-Oriented Prolog

The language SCOOP [VLM88] was developed at University of Montreal as an experimental extension of Prolog. Both classes and objects are modelled through Horn clause databases. All objects of a particular class have a common set of predicates.

Each predicate is either static or dynamic. *Static* predicates are defined in the definition of the class (or its superclasses), and are common to all its objects. These predicates can be viewed as methods. Definitions of *dynamic* predicates may have some initial clauses common to all objects of a class, but they also can obtain new clauses passed via a parameter at object creation and have clauses asserted/retracted during program execution. Dynamic predicate clauses function as local databases of objects, and they can model changeable states (attributes). To improve SCOOP's performance, dynamic clauses are limited to facts, whereas static can be either facts or rules.

Dynamic predicates can be of two kinds—"add" or "replace" ("add" predicates are marked by `+` prefixes in the header of the class definition). If a predicate is "add", the new facts for its definition, which are passed as parameters when creating an object, are added to the facts originating from the class definition. If a predicate is "replace", new facts replace the old ones. In inheritance, contrarily to creation, the subclass clauses of a predicate definition always replace those of a superclass. In SCOOP, cumulation, differently from ESP and POL (see Subsections 3.4.4 and 3.4.8, resp.), happens only while creating an object, not later.

An object in SCOOP is created with a `new/3` predicate, which takes *object-ID*, *class-name*, and *dynamic-clause-list* as arguments. An object can obtain a reference to itself with the primitive `thisobject/1`.

As an example, we define a class `person`, and create an object `Fred`:

```
class person (name/1,child/2,sex/1,+needs/1).
dynamics.
    name(unknown).
    sex(male).
    needs(love).
statics.
    son(S) :- child(S,male).
    daughter(D) :- child(D,female).
end.

new(Fred,person,[name(fred),child(ann,female),
                     child(joe,male),needs(money)]).
```

The local database created for `Fred` will contain the following facts:

```
name(fred).         %replacing name(unknown)
sex(male)           %default setting
needs(love)         %default setting not replaced
needs(money).       %added to the previous default
child(ann,female).
child(joe,male).
```

Message sending is imitated by calls which in the remote case can be of two forms:

receiver: *goal*,

with the obvious semantics, and

receiver `as` *class*: *goal*,

which means that *goal* is attempted on the *receiver* object treated as belonging to *class*. In order to protect locality, the `asserta,` `assertz` and `retract` predicates are forbidden as goals of a remote calls.

SCOOP also facilitates concurrency, and has a discrete simulation capability.

3.4.12 LOCO = LOgic for Complex Objects

The design philosophy of the LOCO [LVV89] language is very different from that admitted in most merging attempts basing on Prolog. The language syntax closely resembles Prolog, but the treatment of the concept of predicate is rather unusual. Besides that, LOCO does not distinguish between objects and classes—both are called objects—, thus having commonality with the classless OO languages.

Objects in LOCO are sets of Horn clauses. As terms, only object names can occur. The *specificity* relation is defined on objects, which is a combination of isa and instance-of relations. One object can be declared a specialization of another either in the definition of the subobject/instance or in that of the superobject/class. In the first case, a type-declaration like syntax is exploited. In the second, a special predicate `_instance/1` is used. In the following example, both `nautilus` and `neptuno` are instances of `hotel`:

```
hotel =
    {name(str).
     nrOfRooms(int).
     street(str).
     city(str).
     address(street=Street,city=City):-
         street(Street), city(City).
     telephone(int).
     _instance(nautilus)};
nautilus =
    {name("Nautilus").
     street("av. de la Piscinas 2").
     city("Bajamar").
     ...};
(hotel) neptuno = {...};
```

As shown in the example, predicate argument places can be named, and in this case, succession of arguments is arbitrary, and omissions are allowed. The example also clarifies why the LOCO philosophy perceives `hotel` like an object, not a class. By `hotel` not a set of all possible or existing hotels (the extent of the hotel concept), but rather a typical, default image of one hotel is meant. This “proto-hotel” can be further refined using the specificity relation.

The `_instance` predicate adds much power to the language. The `_instance` property of an object need not be a fact. It can as well be derived, like in the following example:

```
(hotel) romanHotel =
    {_instance(H):-
         hotel._instance(H),
         H.city("Rome").};
```

where ‘.’ corresponds to message sending. Note that `romanHotel` is defined very similarly to collection formation in CBL (see SubsectionCBL).

Inheritance in LOCO operates on the specificity relation. If a predicate is defined in more than one object along the specificity path, two cases arise. Atoms `p(a)` and `not p(b)` are said to be *inconsistent* iff a=b or if a is an instance of b. Atoms p(a) and p(b) are said to be *conflicting* iff neither of a and b is an instance of the other, nor do they have common instances. In both cases, the atom provable in a more specific object overrules the other. Exceptionally, the so-called *own* predicates like _instance are never inherited.

The above definitions of inconsistency and conflict are quite strange, but they lead us to the LOCO understanding of predicates. A usual predicate in LOCO is "single-valued"—it can be valid for object-tuples among which there must be exactly one which is a specialization of all the others. Thus, a LOCO predicate is like a slot, meant to store an array of values. Nevertheless, the ordinary "set-valued" predicates are also supported by LOCO (distinguished by using braces instead parentheses around arguments), but they are not typical of the LOCO style. The LOCO understanding of predicate enables expression of default and delegation mechanisms.

The authors of LOCO claim that, in their approach to the concept of object, inheritance becomes a special case of delegation. This is even stronger a claim than that by Stein [Ste87] discussed above. As an example of the LOCO delegation, consider:

```
gonzales =
    {name("Gonzales").
     profession("hotelkeeper").
     owns{nautilus}.
     owns{neptuno}.
     ...};
(gonzales) pedro =
    {name("Pedro").
     owns(perdoInn).};
```

pedro inherits everything thats true about gonzales, except for the latter's name which is overruled.

If negation by failure (the not predicate) is supported, a complicated non-monotonicity emerges in LOCO. There exist programs for which there is no unique minimal model etc. The logical theory behind LOCO, the so-called *ordered logic*, has been thoroughly studied in [LV91].

LOCO has an automatic versioning feature (former revisions of objects get a numeric suffix). Also, to objects clauses can be attached that execute at updates (*triggering*, cf. access-orientation).

3.4.13 Objects as Intensions

W. Chen and D. S. Warren made their proposal [CW88], searching for a neat and adequate semantics for dynamic objects. There is no implementation behind their model. It basically deals with objects having dynamic states. Methods, classes, and inheritance are not considered.

Objects are viewed as *intensions*, i.e. they are identified with their histories of states, thus being mappings of time instants to states. (The authors, unfortunately, use a quite confusing different terminology, with the word 'state' applied for time instants, not for tuples of attribute values.)

The underlying language is Prolog. Besides the ordinary static individuals, intensions are introduced (only as variables). Each argument place of each predicate is reserved either for static individuals or for intensions. Logical variables for intensions are marked by a prefix I.

There are two kinds of predicates: *static predicates* and *dynamic predicates.* They are defined in static and dynamic clauses, respectively. Static clauses (indicated by :-) are like ordinary Horn clauses, except that they may contain members for retrieving current states of objects. Dynamic clauses (indicated by ::=) are more like procedures which may update objects, thus shifting the current time instant.

The predicate for intension value retrieval is ::, and it can be used in the form *intens-var*::*static-term*. Since *static-term* may be ground, this predicate can also be exploited for value checks of intensions. The update predicate := is used in the same form as ::, but an invocation of a := goal changes the current time instant into the next, and binds the state of *intens-var* at this new current instant to *static-term*. For other intensions, the *frame assumption* applies: their states will not change, and become what they were at the previous instant.

For intensions, always only the beginning of their history is constrained. The query is always invoked at the instant 0. During deduction, more and more of the histories of intensions becomes constrained. As a result of deduction, the time instant reached at and the bindings of static variables, as well as the bindings at that instant of intensional variables, are returned.

There is an implementational problem about unification of intensional variables. If the beginnings of the histories of two objects coincide, it doesn't necessarily imply that these objects are identical. As a compromise, the deduction procedure adapted disallows unification of two intensional variables when both are constrained. Only when nothing yet is known about one of two intensional variables, they can be unified. This convention does not appear to be limiting, since in OO systems usually anyway only current states of objects are kept.

Consider how a stack can be implemented as an intension.

```
isempty(IStack) :- IStack::[].
top(IStack,El) :- IStack::[El|Rest].
push(IStack,El) ::= IStack::Curr, IStack:=[El|Curr].
pop(IStack,El) ::= IStack::[El|Rest], IStack:=Rest.
```

The proposal has a clear semantics in intensional logic.

3.4.14 Object Clauses

The proposal [Con88] by J. S. Conery of Oregon University, OR, USA, in many respects resembles that of the previous section. Its main objective is to cope with dynamic states. The base language is Prolog. Though methods and classes are considered, inheritance still is not discussed, and it is not obvious how to incorporate it into the model. The idea that it is the deduction procedure, who should automatically shift the current time instant, has been preserved. Objects themselves, though, are modelled not via individuals, but via predicates in this solution. Another difference is that semantically an object is characterized only via its current state, not via its whole history.

There are two kinds of predicates in the model. A *procedural predicate* is normal Prolog predicate. An *object predicate* stands for a class. It has got argument places for an instance name (the first argument, by convention) and for attribute values. Procedure literals have the usual interpretation. As the head of a clause, a procedure literal indicates to a definition case of the procedure, and in the body, it is a call to the procedure. Object literals can be explained as follows. A positive object literal means that there exists an object of the predicate-determined class with the arguments-defined name and attributes at the current time instant. A negative object literal means that such an object will exist at the next instant. A clause where a given object predicate (with a given instance name argument) occurs in the head, but not in the body, means destruction of this object. Otherwise, the object is transformed.

To control the order and timing of object transformations, *object clauses* are introduced. Their head is a conjunction of a procedure literal and an object literal. For pure logic, this would be only a syntactic sugar, because such a "clause" would be equivalent to two ordinary clauses.

In the query and in the clause bodies, procedure literals must precede object literals, and this is to be maintained during resolution. The resolution algorithm is the following.

- If the leftmost goal is a procedure literal, it is removed from the current goal statement, and the system finds a clause with a matching head. If this is a Horn clause, its body is merged into the current goal statement. If it is an object clause, the system must find an object literal somewhere in the right hand side of the current goal statement, that unifies with the head object literal of that object clause. The object literal found is then removed from the current goal statement, and the body of the clause is merged into it.
- If the leftmost goal in the current goal statement is an object literal (i.e. there are no procedure predicates left), it is removed from there, and the system has to find a Horn clause with a matching head, and merge it into the current goal statement.

The following activities associated with a class can be imitated in the model:

- *Validation procedure* is a set of Horn clauses where each head is an object literal. These clauses describe all the valid objects of the class.
- *Creation procedure* is a set of Horn clauses where each head is a procedure literal, and each body contains at least one object literal with the predicate symbol corresponding to the class.
- *Method* is a set of object clauses where the predicate of all the head object literals is the one corresponding to the class, and the head procedure literals all have the same predicate.

Encapsulation can be provided by forbidding object head literals elsewhere than in class definitions. If this is ensured by a preprocessor, the only way to create an object, is to call a creation procedure, and the only way to access or modify an object, is to call a method.

The three components of a class are used at different phases of computation. The query should be a set of procedure literals. Some of its goals lead to object creations. Next, objects are transformed by methods, and some more objects may be created, until no procedure literals remain in the goal statement. Last, the validation procedures check the "legitimacy" of objects and simultaneously destroy them, finally reaching at empty resolvent.

Consider an implementation of the class of integer stacks:

Validation procedure :
```
    stack(ID,[]).
    stack(ID,[El|Rest]) ← integer(El) ∧ stack(ID,Rest).
```

Instantiation procedure :

```
new_stack(ID) ← stack(ID,[]).
```

Methods :

```
isempty(ID) ∧ stack(ID,[]) ← stack(ID,[]).
top(El,ID) ∧ stack(ID,[X|Rest]) ← stack(ID,[X|Rest]).
push(El,ID) ∧ stack(ID,Curr) ← integer(El) ∧ stack(ID,[El|Curr]).
pop(X,ID) ∧ stack(ID,[El|Rest]) ← stack(ID,Rest).
```

Since in all method definitions, the action on an object is essentially implemented through re-creation of the object, the predicate of the object must always be repeated in the body of the object clause, even if no state change is involved. This is, of course, an effect of the uniformity of the model, but practically, the lack for the default of "no changes" causes some inconvenience.

The model facilitates anonymous method calls. Another remarkable feature is that if an object is sent a message, and it returns a result that cannot be used, then if the method definition consists of several clauses, the system backtracks, and the object tries to handle the message in a different way. In this, turnbacks in time are involved, and so time is reversible in a sense!!

3.4.15 ObjVProlog

The design of the ObjVProlog [MLV89] language has been guided by the example of ObjVLisp model, which provides metaclasses and handles metaclass/class/instance concepts in a systematic and reflexive way. It was proposed by the same research group that earlier worked out SCOOP (see Subsection 3.4.11). In SCOOP, classes were statically defined and were not objects, and metaclasses were missing. In ObjVProlog, all classes are objects, and all metaclasses are classes (and therefore, also objects).

ObjVProlog, like ObjVLisp, relies on two classes: `class` and `object`. `object` is a superclass to all classes, and does not have any superclass itself. It is the root of the *inheritance graph*. `class` is the first object of the system, and an instance of itself, thus being the root of the *instantiation graph*. It is also the first metaclass. `class` is manually created, whereas `object` is created from `class` automatically thereafter.

Every object has an attribute `isit` described in `object`, which stores the name of its instantiation class. Each class, being an object, also has the same attribute, storing the name of its metaclass. Its other attributes `name, supers, i_v, dynamics`, and `statics` (their descriptions, to be more exact), and its method `new`, are inherited from `class` already.

Like in SCOOP, objects in ObjVProlog are Horn clause databases. These clauses implement the state of objects, which in ObjVProlog is modifiable through `assume` and `forget` predicates (semantically "cleaner" versions of `assert` and `retract`, respectively). The inheritance mode is overriding, and relies on `supers` to access the superclasses' shadowed definitions. Multiple inheritance is provided using CLOS-like linear extensions.

The following example illustrates the elegant uniformity of treating metaclass/class/instance concepts in ObjVProlog:

```
class::new(point,
   [[name(point)],
    [supers([object])],
    [dynamics([x/1,y/1])],
        [statics([
            [(replace_x(NX) :- ...)],
            [(replace_y(NY) :- ...)],
            [(initialize(...) :- ...)],
            [(display :- ...)]
          ])]]).
    point::new(P2,[[x(10)],[y(20)]]).
    P2::display.
```

`::` is used as message sending operator here. First, a class `point` as a subclass of `object`, and then an instance `P2` of `point` are created. Finally, P2 is displayed.

3.4.16 Prolog++

Prolog++ [Mos90] was developed by Phil Vasey at LPA, and is a layer on top of Prolog. Prolog++ is a classless language, and what it terms as inheritance, is like delegation (cf. LOCO, Subsection 3.4.12). Objects cannot be created dynamically, and their attributes cannot change—their values are fixed forever, and can just be computed if stored implicitly. Quite unusually, an attribute can have a number of values at a time, and it can also have arguments.

An object definition is a set of clauses, braced by the `open_object` and `close_object` keywords followed by the object name. Note that these keywords do not denote any action (`close` does not mean destruction!), but serve as delimiters only. The heads of the clauses for attributes are of the form

attribute`=`*value*,

where both *attribute* and *value* are terms. Since methods are modelled by predicates, the heads of the clauses for methods are literals acting as patterns of calls. Both types of clauses can be either facts or rules. Hence even the values of attributes can be set conditionally. In the bodies of these clauses, attributes and methods of other objects can be called. The syntax for attribute calls is *receiver*`::`*attribute*, and for methods it is *receiver*`<-`*message*.

In support of encapsulation, attributes can be declared private to their object. The declaration in the form

`private` *attribute*`/`*arity*.

must appear before the definition of the attribute.

Each object can have a number of superobjects (prototypes) whose names are stored in its special attribute `super`. If an attribute or a method is not explicitly defined for a given object, the closest definition upwards in the object/superobject hierachy is appropriated. The definition clauses do not cumulate. An object variable `self` can be used in attribute calls and method calls as a substitute for the receiver. As the receiver can also be logical variable, intensional calls are enabled.

Consider a family relationships example.

```
    open_object human.
        father=self::eldestSibling::father.
        private eldestSibling/1.
        eldestSibling=Eldest :-
            self::olderSibling=Elder ->
                Elder::eldestSibling=Eldest
    ;   Eldest=self.
        married :- self::spouse=_.
        sibling(older)=X :-
            self::olderSibling=Y,
            (Y::sibling(older)=X ; X=Y).
        sibling(younger)=X :-
            self::youngerSibling=Y,
            (X=Y ; Y::sibling(younger)=X).
        brother=X :-
            self::sibling(_)=X,
            X<-male.
        age is general::thisYear - self::birth.
    close_object human.

    open_object male.
        super=human.
        spouse=self::wife.
        retirementAge=65.
        sex=male.
        male.
    close_object male.

    open_object chris.
        super=male.
        name='Chris Moss'.
        height=183.
        birth=1944.
        wife=karen.
        olderSibling=jane.
    close_object chris.
```

In this program fragment, the father of a family is stored only in the object for the eldest of children. For each person, only his next older and next younger siblings are stored in his/her object. Data about the father and other siblings can be derived through methods. Since `eldestSibling` was introduced only for deriving `father`, it was made private for `human`.

To provide dynamics, Prolog++ makes use of *instances.* ('Object' and 'instance' are *not* synonymous in Prolog++). Prolog++ instances can be created and deleted dynamically from Prolog++ objects in the same way as objects from classes in class-based OO languages. Besides static attributes, instances can have dynamic attributes, described (declared) in the object definition. Dynamic attributes act more like traditional OO object attributes. A dynamic attribute must be an atom, and cannot be a general term. It can only have a single value, though repeatedly changeable by assignments. Its value is computed only on assignments, not every time it is retrieved. It cannot have any associated conditionals (but a definition clause of a static attribute may have a body).

Instance creation and deletion are methods of a special object `instance`. To create an instance, one must use a goal

    instance <- new(*object,instance*).

Deletion is achieved through a goal

```
    instance <- delete(instance).
```

The literal for assignment has a form *attribute*`:=`*value*. For retrievals, the ordinary `=` is used.

It is curious that both static attributes and methods have been implemented in Prolog++, since having only one of the two would have sufficed. It is merely a matter of convention whether to write

```
    smth(term1,...,termN)=term
```

or

```
    smth(term1,...,termN,term).
```

In the first expression, `smth` is a multi-valued function, in the second, it is a predicate determining the graph of this function. Cf. also Vulcan (Subsection 3.4.10) where immutable objects could be represented as terms instead of perpetual processes (the latter were the only internal representation for mutable objects in that language).

3.5 Hybrid Mergers

In some solutions, no evident bias towards one of the two paradigms under merging can be detected. These approaches usually rely on advanced generalizations of some characteristics of the "source" paradigms. A selection of them is reviewed below.

3.5.1 ALF = Alltalk Logic Facility

ALF [Mel88] has been developed to offer a Prolog-like model for logic programming to the users of Alltalk, a system which provides persistence to Smalltalk objects without adding new language syntax. Since ALF itself has been implemented in Alltalk, ALF also provides persistence for its specific objects, i.e. for rules, facts and queries.

ALF draws heavily on LOGIN [AN86], a logic programming language with built-in inheritance. LOGIN generalizes unification, putting it to take into account the lattice relationship among types which closely corresponds to the OO class hierarchy. However, LOGIN is more complicated than ALF, which, due to its origin in Smalltalk, only has single inheritance. In LOGIN, also the syntax of predicates and terms has been expanded, allowing "attribute labels" (names for argument places), which is close to Smalltalk, where attributes have names. (Cf. also LOCO, Subsection 3.4.12). In ALF, an object is represented as a compound term with the functor as the object name, and the arguments for the attribute values (cf. Vulcan, Prolog++, Subsections 3.4.10, 3.4.16, resp.).

An example of ALF syntax is:

```
    Hearty(thing=X:) <-
        Healthy(thing=X:Person(profile=Profile(age=W:,country=Y:,
                            hobby=Sport(name="jogging",level=Z:)))),
        LessThan(smaller=W:,larger=65),
        SportsLoving(Y:),
        LevelLessThan(lower="novice",higher=Z:).
```

The argument places of predicates are named, so that only the relevant arguments need be supplied, and in an arbitrary order. Logical variables (marked by : suffixes) can be "classed", and class qualification can be nested indefinitely. The above clause states that any person who is healthy, less than 65 years old, comes from a sports-loving country, and has a hobby of jogging with an expertise level greater than "novice" is hearty.

Unification is accomplished through a special method in the class `Object`. Rules, facts, and queries are instances of the class `Clause`, whilst predicates are grouped into the class `Predicate`. To send Smalltalk messages from ALF programs, the predicates `SendN` have been built in as subclasses of `Predicate`. These predicates take arguments `receiver, answer, selector` and parameters. There is also a built-in `Exists` in ALF predicate. This answers `true`, if its single argument `is` exists in the database.

ALF provides a librarying facility. Clauses in ALF are grouped into AlfPrograms. Instances of the class `AlfProgram` have instance attributes `ruleDictionary, author, date, comment`, and `name`. The class attribute `pgmDictionary` of `AlfProgram` registers all AlfPrograms in the system, keyed by `name`.

3.5.2 PRIZ and NUT

The programming environments PRIZ and NUT, developed at the Institute of Cybernetics, Estonian Academy of Sciences [MT90], differ greatly from all solutions considered above. Both the treatment of logic and OO are non-traditional. Logic programming is understood in a broad sense as the use of constructive proofs for building correct programs, and not merely as predicate Horn clause programming. Programs in PRIZ and NUT are specification-level rather than algorithms, and the logic lies in planning, not in execution as in Prolog-like languages. The logic language underlying these systems is low-level (in contrast to Prolog, for instance), and OO is used to provide a better input language.

Horn clause programming has a clear semantics, but it cannot be maintained, or it is hard to maintain it in practice. To overcome the deficiencies of the depth-first search and the circumstance that variables can't be varied etc., special cut and assert/retract predicates and other techniques are usually utilized. However, these do not have neat logical semantics. Roughly, Horn clause resolution means deriving the formula

$$\forall\overline{x}(P(\overline{x}) \to \exists\overline{y}R(\overline{x},\overline{y})),$$

where $\overline{x}$ are input variables, $\overline{y}$ are output variables, P is a check for input variables, and R is an input-output relation.

In some cases it is possible (and for planning purposes it almost always is) to make R dependent on $\overline{y}$ only. Then the above formula reduces to

$$\forall\overline{x}(P(\overline{x}) \to \exists\overline{y}R(\overline{x})),$$

which is equivalent to

$$\exists\overline{x}P(\overline{x}) \to \exists\overline{y}R(\overline{x},\overline{y}).$$

Now both the antecedent and consequent of the implication are closed formulae, and can be treated as propositional constants:

$$P \to R.$$

The language of PRIZ is actually a bit more complicated, and the general form of a formula is:

$$\overline{(\overline{U} \to V)}\&\overline{X} \to Y.$$

Overlines stand for conjunctions over vectors. The latter formula can be read as, "Y can be computed from $\overline{X}$ and (functions realizing) $\overline{(\overline{U} \to V)}$". It turns out that nesting over depth 2 can always be avoided by introducing new propositional constants.

It has been proved that there exists a complete proof search procedure for the above-type formulae. Besides that, the proof tree—which actually proves nothing more than that a solution exists—will always "contain" the very algorithm to find that solution (the λ-term corresponding to the tree is exactly what is needed). The decribed approach to program synthesis is called *deductive* by the authors, and the calculus used is called *structural synthesis rules.* The kind of logic programming used is called *propositional logic programming*, to emphasize the contrast to Horn clause one, which is predicative. In propositional logic programming, first a *planner* constructs the contours of the algorithm (using logic), and consequently, a *solver* computes the solution (either mechanically or using tools different from logic).

To provide a user-friendly front end, an OO style input language is provided. No distinction between classes and objects is made, and single inheritance has been provided.

A "program" is a list of specifications. A specification is written in the form

object: *type-specifier*,

where *type-specifier* can be one of the following:

1. a primitive type: numeric or text,
2. a name of an object already specified,
3. a structure of the form $(x_1 : t_1; \ldots; x_k : t_k[; relations])$ where $(x_1 : t_1; ...; x_k : t_k$ are specifications, and *relations* are either equations of the form $E_1(x_1, \ldots, x_k) = E_2(x_1, \ldots, x_k)$, where E_1 and E_2 are arithmetical expressions, or references to pre-programmed functions in the form $x_{i_1}, \ldots, x_{i_m} \to x_i\{f\}$, where f is the function name.

In specifications, components of other objects can be referred to using the dot-notation. This slightly corresponds to message sending.

The solver can only explicate a variable from an equation, if the values of all the other variables in that equation have become known. The solver cannot solve equation systems in the strong sense. The idea of equation solving is close to constraint-orientation [BS86].

The specification is translated into formulae, and so is the query. The planner has to derive the query formula from the specification ones.

The example

```
rectangle:
    (a,b,s,p,d:numeric
     s=a*b
     p=2*(a+b)
     d=sqrt(a^2+b^2))
square:rectangle a=b
thissquare:square s=25
```

will yield the following formulae:

```
rectangle -> rectangle.a
rectangle -> rectangle.b
rectangle -> rectangle.s
rectangle -> rectangle.p
rectangle -> rectangle.d
rectangle.a & rectangle.b & rectangle.s &
    rectangle.p & rectangle.d -> rectangle
    /* rectangle and its components */
rectangle.s & rectangle.a -> rectangle.b
rectangle.s & rectangle.b -> rectangle.a
rectangle.a & rectangle.b -> rectangle.s
    /* the equation s=a*b */
...
    /* the other equations similarly produce formulae */
square -> square.a
square -> square.b
...
    /* all the above axioms about rectangle hold
       also with square */
square.a -> square.b
square.b -> square.a
    /* the equation a=b */
thissquare -> thissquare.a
thissquare -> thissquare.b
...
    /* all the above axioms about square hold also
       with thissquare */
thissquare.s
    /* the equation s=25 */
```

If now, a query `->thissquare.a` is set forth, the planner will first analytically find the algorithm, and then the numerical solution `a=5` will be computed. If `square.a->square.s` queried, no numerical solution is possible, and the analytic solution `s=a^2` will be synthesized.

NUT is an advanced, more object-oriented version of PRIZ.

3.5.3 FOOPlog

The FOOPlog proposal [GM87], similarly to PRIZ and NUT, takes a non-traditional standpoint about logic. Usually, functionality, logic, and OO are viewed as (the) three orthogonal paradigms. The authors of FOOPlog claim, on the contrary, that actually, only two orthogonal paradigms—logical and OO—exist. They take a broad view on the logic paradigm, stating that a logic language is one whose programs consist of sentences in a well-understood logical system, whose operational semantics is deduction in that system, and whose denotational semantics is given by a class of certain algebraic models. After that they observe that both the functional and conventional logic (i.e. Horn clause) paradigms are logical in their sense—the functional paradigm relies on a kind of equational logic, and the conventional logic paradigm relies on first-order Horn clause logic. Finally, to avoid the emerging collision, they rename the Horn clause logic paradigm into relational paradigm.

The FOOPlog proposal pretends to merge all the above-mentioned three paradigms: functional, object-oriented, and relational. To fulfil the task, a hierarchy of languages is introduced: OBJ serves as a basic functional language, FOOPS integrates functionality and OO, Eqlog integrates functionality and relations, and FOOPlog integrates all three. The authors distinguish between functional-level and object-level data. The former are structured via sorts, the latter via classes.

On the functional level, functions, and on the object level, methods are defined via equations. Unlike to PRIZ and NUT, equations in OBJ and its "successors" can be applied only left-to-right, acting as rewrite rules, progressively transforming terms to reduced forms. One may think of these equations as of explicit function definitions (possibly involving recursion scheme etc.), in contrast to implicit definitions (constraints) of PRIZ and NUT. On the object level, equations defining a method are written by indicating its effect on each slot (attribute). To simplify the notation, equations merely stating that a particular method does not affect particular slots, can be omitted.

The operational semantics on both levels is rewriting. The denotational semantics is formulated in terms of *initial algebras*. Given an order-sorted signature Σ and a set of equations E, the emerging initial algebra can be intuitively characterized by the two following conditions:

no junk: every element of the algebra is denoted by some Σ-term;

no confusion: two Σ-terms denote the same element if their equality can be proved, using the equations E as axioms.

Programs consist of modules. Function-level modules essentially define *abstract data types*, whereas object-level modules *abstract machines*. An abstract machine is an equivalence class of behaviorally equivalent algebras.

FOOPS introduces a novel error-handling technique. For every user-defined sort `S`, an error supersort `S?` is automatically created to contain the error messages associated with the sort `S`. Errors are therefore handled as normal data of error supersorts, and this in a sense legalizes error situations. An analogous technique is also utilized with classes, where error superclasses are associated with user-defined classes.

In [GM87] FOOPS is discussed thoroughly, mainly from the perspective of the ideas on ADT's and abstract machines, and also on reflection. The concrete details of FOOPlog (which would be our main interest), the interaction of OO and relations in this language, are not considered in that paper, unfortunately.

3.5.4 Maude

The language Maude [Mes90] is a further development on the line of OBJ, FOOPS, and FOOPlog. A solution to the dilemma between timeless, static understanding of logic, and dynamic aspects of computation, such as input-output, concurrency, changing of values (or, more generally, changing of database), has been proposed in a form of *rewriting logic*.

As before, logicality in programming is understood in a wide sense. Nonetheless, what formerly was called equational logic, has now (possibly due to developments in logic and automated proving) openly and with a sound justification been renamed into rewriting. Indeed, equations (or constraints) are only axioms for a certain static algebraic structure, not containing too much hints on how to search derivations (i.e. on how to compute actually). Rewriting logic, on the contrary, formalizes the very way how to search for derivations, and covers the dynamic aspect, thus being a sort of metatheory over the former.

Though very interesting and outstanding in its originality, the Maude philosophy is distanciated from the general understanding of the logic paradigm, maintained in this thesis, and is not considered in more detail here.

3.6 Comparison

The solutions considered vary widely, since different authors have proceeded from different starting-points, and have kept in mind different purposes. Their views on the OO and logic paradigms do not coincide, and neither do their objectives for merging do.

A merging objective is a "weighed sum" of two controversial strives towards *convenience* and towards *uniformity*. Aiming at convenience (i.e. at ease of expressibility) tends to lead to a solution where many techniques are available, but have been incorporated in an ad hoc way. Strive for uniformity, vice versa, may set limits on the amount of features that can be captured.

The combinations where logical features are introduced into OO languages, typically have been designed to be convenient. Most frequently, they enrich an OO language with means for deductive retrieval, providing constructs to perform backtrackable unification and resolution. Expressibility support was also the main objective behind the Prolog/Loops language interface.

The combinations where a logic language serves as a basis, and OO features have been added, mostly put more emphasis on uniformity. Some of them capture only few OO concepts and techniques, but give them a neat logical basis (Prolog/KR, Objects as intensions, Object clauses). Characteristically, in many cases, modifications have been made to the classic Horn clause logic programming. E.g. "single-valuedness" is the norm for LOCO predicates; in solutions where logical variables have been used for object representation, they have been given a wider scope than just one clause etc. Thus many of those approaches actually embody a search for logical foundations of OO.

Hybrid solutions are the strongest devotées to uniformity. They view both the OO and logic paradigms extremely flexibly, being eager to vastly modify both of them for the sake of uniformity.

From among the solutions where logical features were introduced into and OO language, Orient84/K and KSL/logic make use of logic programming to get the merits of the deductive retrieval mechanism readily present there (the methods `unify` (`foreach_unify`), and `ForAll`, respectively). KSL/Logic goes further than Orient84/K, viewing logic constructs as objects. This choice is unique to KSL/Logic. CBL can be viewed from two aspects. One might simply say that it merely offers a systematic way of writing complex conditionals in definitions. But equally one might claim that the similarity of the style used in its conditional definitions to the logic programming style is striking. Still, not the full power of the logic programming unification is applied in CBL, it only appears in matching of factual and fictive parameters at definition calls. Also, clauses do not consist of truth-valued members only, all LISP terms are allowed as members, and so it is more like some functional language where conditionals are easily expressible.

A comparison of solutions where OO features have been introduced into logic languages, is presented in Tables 3.1 and 3.2. Most of the solutions are class-based, and rely on class inheritance. Prolog/KR, CPU and LOCO do not distinguish between classes and objects, and inheritance in them is delegation-like. The other major differences of these solutions can be pointed out, studying how objects and classes are represented, how state changes are modelled, and how inheritance is understood. Methods are essentially represented via predicates, and defined as predicates usually are in logic programming, i.e. by means of lists of clauses. Only Chen's and Warren's 'Objects as intensions' do not talk about methods, but actions on objects, nevertheless, are performed via resolving goals with clauses.

Representation of objects and classes

As the method definitions are stored in lists of clauses, it is clear that objects and classes must be represented in a way that enables them to refer to sets of clauses. In nearly all the solutions, both objects and classes are represented as atoms (NB! we use Prolog terminology here, meaning 'individual constant', not 'atomic formula', when saying 'atom'). In some solutions (Mandala, POL, CPU, Conery's proposal) clauses of method definitions are somehow marked (e.g. prefixed) by such atoms. Most solutions, however, collect all the method definitions owned by a given class (or by a given object, in classless systems) as well as other related information (attribute descriptions/values, supers etc.) into a class (object) definition. In those, the atom standing for the class (object) appears in the header of its definition.

The proposal by Zaniolo does not allow dynamics. Since object states cannot change in this system, an object can be identified by its state. This enabled Zaniolo to represent classes by predicates with argument places for state attributes, and objects by literals with ground arguments. In Conery's proposal, classes similarly are represented by predicates, but since objects are dynamic there, these predicates have an additional argument place for the object atom-ID.

In Mandala, objects are represented by literals (perpetual processes) with arguments for state attributes, class atom-ID, and streams. Differently from Conery's proposal where object predicate interpretations changed in time, Mandala relies on the usual static logic, and streams are used to distinguish between object state revisions. The circumstance that both in Mandala and Conery's proposal instance and class representations are one argument of another, makes instance/class links dynamic in them.

In Vulcan, SCOOP, 'Objects as Intensions', and ObjVProlog, the atoms to represent objects are generated by the system in order to guarantee their uniqueness. In those languages, the user-end representation of an object is a logical variable.

In some of the solutions where objects are represented by atoms, anonymous (intensional) messages are allowed. In writing such messages, the receiver object is left open by indicating an uninstantiated logical variable as its ID. In [LM88], it is pointed out that Zaniolo's proposal essentially supports a variant of intensionality. In that language, a receiver can be a partially specified object of a given class, represented by a literal with containing uninstantiated variables in its argument terms.

Modelling state changes

Zaniolo's approach is fully static, and so is Prolog++ on its objects' level. ESP, LOOKS, POKRS, SPOOL and Prolog++ (on the instances' level) do not attempt to give state dynamics any logical interpretation, and make use of *imperative assignments.* Attributes in those languages are represented by ("illogical") variables similar to those used in procedural languages.

Among more logical approaches to tretaing state changes, two are the most wide-spread. In [LM88] they are called *procedural states* and *declarative states*, respectively.

- Predominantly, states are revised using the `assert` and `retract` built-in predicates.

Language	Base language	Obj. repr.	Class repr.	Instance creation	Inst./Class rel. decl.	Cl./Supercl. hier.		Metacl. repr.	Cl./Mtcl. hier. decl.	Other hier's
						kind	decl.			
Zaniolo	Prolog	literal	pred.	—	lit./pred.	mult.	sep. fact	—		—
Prol./KR	Prolog	(*world*) atom		—	—	dyn., single	nesting	—		—
Mandala	KL1 (CP)	perp. process	(*world*) atom	—	dyn., class ID is an arg. of the obj. process	mult.	sep. fact.	(*manager* object)	fixed autom.	*part/ whole*, decl'd in sep. facts
ESP	KL0 (Prol.)	atom	atom	?		mult.	in the cl. def.	—		—
LOOKS, POKRS	Prolog	atom	atom	send msg. `new` to class	dtm'd by creation	mult.	in the cl. def.	atom	in the cl. def	—
SPOOL	Prolog	atom	atom	send msg. `new` to the class	dtm'd by creation, or sep. fact	mult.	in the cl. def.	atom	in the cl. def.	—
POL	Prolog	atom	atom	—	sep. fact	mult.	sep. fact	—		—
CPU	Prolog	(*unit*) atom		—	—	ctrl'd via metaprg.		(*meta-unit*)	fixed autom.	—
Vulcan	CP	log-var	atom	call `make`	dtm'd by creation	mult.	in the cl. def.	—		—
SCOOP	Prolog	log-var	atom	call `new`	dtm'd by creation	single	in the cl. def	—		—
LOCO	Prolog	(*object*) atom		—	—	mult.	in the obj. or super-obj. def.	—		—
Intensions	Prolog	intens-log-var	—	—	—	—		—		—
Conery	Prolog	atom	*obj. pred.*	call a creat. proced. def'd in sep. clauses	dyn.; obj. ID is an arg of the cl. *obj. pred.*	—		—		—
ObjVProl.	Prolog	log-var	atom	send msg. `new` to the class	dtm'd by creation	mult.	in the cl. def.	atom	dtm'd by cl. creation	dyn. *part/ whole*, stored in *dyn. pred's*
Prolog++	Prolog	(*object*) atom	—	—	—	mult.	in the obj. def. via an attr.	—	—	—
		(*instance*) atom	(*object*)	send msg. `new` to a spec. obj. `instance`	dtm'd by creation	—		—		

Table 3.1: Comparative table on integrations of OO into logic languages (beginning)

Lang.	State attribute						Method			
	repr.	descr.	value changes	default value	external access	spec. char's	repr.	def.	extra features	inheritance modes
Zaniolo	arg. places of the obj. lit.		stat.	—	open	—	pred.	single clause in the class def.	—	overr.
Pr./KR	assoc'd pred's def'd by clauses in the world def.; dyn. provided through `ass./retr.`									cumul. (`assert`); overr. (`define`)
Mand.	arg. places of the obj. lit.		dyn.; streams	—	open	—	pred.	sep. clauses	—	overr.
ESP	ID	in the cl. def.	dyn.; assgm's	assgm. in the cl. def. if.	rem. calls	—	pred.	clauses in the cl. def.	—	cumul.; overr. (`!`, demons)
LOOKS, POKRS	ID	in the cl. def.	dyn.; assgm's	assgm. in the cl. def.	via methods	—	ID	ID:*arg's*: *body* in the cl. def.	—	overr.
SPOOL	ID	in the cl. def.	dyn.; assgm's	—	via methods	—	pred.	clauses in the cl. def.	anon. msg's	overr.
POL	facts about the obj's ID, static				open	multi-valued	pred.	sep. clauses	anon. msg's	cumul.; overr. (`withdef.`, `withdtm.`)
CPU	assoc'd pred's def'd by clauses in the unit def.; dyn. attr's repr'd by pred. arg. places may be provided through recursive pred's (assoc'd to *instances*)									ctrl'd via metaprg.
Vulcan	log-var	in the cl. def.	dyn.; var's for up-dates pre-fixed by `new`	—	via methods	—	term	sep. "clauses"	—	overr.
SCOOP	*dyn. pred.*	in the cl. def.	dyn.; `ass./retr.`	facts in the cl. def.	rem. calls	multi-valued	*stat. pred.*	clauses in the cl. def.	—	overr.
LOCO	assoc'd (generally "single-valued") pred's def'd by clauses in the obj. def.; dyn. provided through `replace`; versioning; anon. msg's									overr.
Intens.	value of the obj. ID		dyn.; step-wise constraining	—	open	history of states	—			
Conery	arg. places of the obj. pred.		dyn.; obj. re-creation	—	via methods	vali-dation	*proced. pred.*	sep. clauses	anon. msg's	—
ObjVPr.	*dyn. pred.*	in the cl. def.	dyn.; `assume/forget`	facts in the cl. def.	rem. calls	multi-valued	*stat. pred.*	clauses in the cl. def.	—	overr.
Prol.++	term	in the obj. def.	stat.	facts in the obj. def.	rem. calls; meth's if priv.	multiv'd, can have arg's	pred.	clauses in the cl. def.	anon. msg's	overr.
	instance has the properties of the obj., plus may have dyn. attr's									
	ID	in the obj. def.	dyn.; assgm's	facts in the obj. def.	rem. calls; meth's if priv.	—	—			

Table 3.2: Comparative table on integrations of OO into logic languages (end)

These predicates and most of their variants do not have neat logical semantics. In ObjVProlog, slightly cleaner predicates `assume` and `forget` are made use of.

- Mandala and Vulcan (on its underlying Concurrent Prolog level) exploit streams. These lists of yet unprocessed messages (also called communication channels) serve as revision identifiers when occurring among predicate arguments.

  The philosophy of the CPU dynamic instances also is close to that of the stream-based solutions (recursion, communication channels).

In 'Objects as Intensions', a history of states fully specifies an object. Thus an object is essentially an infinite list of states. Therefore, the logical mechanism behind 'Objects as Intensions' needs no dynamics at all. During execution, state lists just become more and more instantiated.

In Conery's proposal, special so-called object predicates are used to capture dynamics. Their interpretation changes in time. Time transitions are committed at resolutions with object clauses.

The choice in favor of variants of `assert` and `retract` allows the concept of attribute to be understood more generally. Since attributes in such cases are represented by predicates, a "value" of an attribute need not be a single individual, most generally it is a function whose range consists of sets of tuples of individuals. The only remaining difference between attributes and methods in such an approach is that methods can be defined only once and forever, whereas attribute values can always be redefined.

The similarity between attributes and methods is so strong when both are represented by predicates, that some solutions of the just-commented kind allow "messages" that "invoke" attributes from outside. In other words, remote calls can be made to attributes to retrieve their values. This, of course, is not adhering to the principle of encapsulation, but it is still better than no protection, since such a discipline usually at least prevents changing values from outside (attributes are read-only for outside objects).

Backtracking restores past states in 'Objects as Intensions', in Conery's proposal, and in ObjVProlog. The effects of imperative assignments and plain `asserts-retracts` are not removed at backtracking.

Understanding inheritance

In usual OO systems, if a class (or object) owns a method definition, this definition *overrides* any definitions of the same method from above in the class/superclass (object/prototype hierarchy). In the context of logic this means that if its given invocation failed when it was attempted to resolve the call goal with the definition clauses owned by a class (object), no definition clauses from above are searched for or tried. This kind of inheritance is called *call/return* in [Gal86]. Most solutions adapt this approach to inheritance.

In procedural OO languages, concatenation of method definitions might not make sense. It would yield a new method with its invocation equivalent to serial invocation of the original ones. But in logic programming, invocation of such a concatenation would mean that if resolution with the clauses from the first definition failed, the clauses from the second definition are attempted, and if they fail, the third definition is tried etc. Thus it may turn out reasonable to let method definitions *cumulate* down the hierarchy. This is what is called the *success/failure* understanding of inheritance in [Gal86]. The ultimate purpose of a method invocation according to it is not to get the invocation result somehow, but to get it result successfully, if possible.

In Prolog/KR, ESP, and POL, cumulation is the default inheritance mode. Overriding can be achieved, if desired, using special language constructs. The simplest of them—cut—is readily present in Prolog.

In CPU, there are no preprogrammed inheritance modes. Inheritance semantics can be flexibly defined in metaunits via metaprogramming.

In logical terms, cumulation can be termed as *monotonic* (adding new axioms can only increase the set of theorems). Overriding is *non-monotonic* (adding new axioms may refute some of the former theorems).

Concerning the hybrid solutions, we merely make a few remarks at the moment.

ALF is similar to KSL/Logic in that logical constructs are objects, and in that unification is accomplished through a special method. In addition to this, it is noteworthy that, from the logic view-point, objects are represented by individuals denoted by compound terms with functors standing for classes and argument places meant for attribute values. The unification mechanism used is very interesting.

In PRIZ, two layers can be clearly distinguished. The internal layer is logical and does not pretend to suit for comfortable OO programming. The external user-end layer, on the contrary, takes cares of OO-style expression, but does not aspire for leaving an impression of high logicality on a user.

Chapter 4

Towards a New Solution: Kernel Ideas and Logical Basis

This chapter contains a proposal towards a new merger. The proposal has been inspired by a belief that the two merging objectives ideals that we referred to in Section 3.6—convenience and uniformity—are badly compatible. To overcome the difficulty, Section 4.1 suggests a two-layer architecture of a merger system. The rest of the chapter is an outline of the internal (logical) layer whom we choose to base on modal logic. It is organized as follows.

Section 4.2 discusses developments towards structurality within the logic paradigm. Section 4.3 argues that there are two dimensions of evolution in OO. Then, in Section 4.4, the exact functions of each of the two layers are fixed. This section might be viewed as an attempt to extract from the OO principles, concepts, and techniques those having natural logical counterparts. Section 4.5 serves as an introduction to (general) modal logic. Sections 4.6–4.9, elaborate the concrete modal logics for the internal layer. Section 4.10 discusses the pragmatics of the designed logics, i.e. their application as programming languages. Section 4.11 illustrates the created formalism by some examples, and evaluates it. Finally,

4.1 Two-Layer Architecture

One of the reasons for the confrontation of convenience and uniformity in mergers lies in the very nucleus of the OO and logic paradigms. OO tries to ease the software development process and offers many shortcuts, and it was "invented" by practical people. Theoretical explanations to the mechanisms behind OO techniques have nearly always been given after these techniques having found their way to practice already.

The logic paradigm emerged in a circle of more theoretical people, and it originally cares more about uniformity than about being suitable for large practical applications. The logic paradigm community has been concerned about clear semantics, and the "deviation" of Prolog constructs like `!`, `is`, `assert`, `retract` from the "logicality ideal" has often been worried about. Most of modifications that might give them a logical blend, bring along increased complexity. Despite this, for the sake of uniformity, these modifications sometimes still are used.

Such ideals are seldom fully appreciated by practical people. The authors of a combination of ADT's, logic programming, and databases [WWE89], for instance, write, "...Also, it would be better if operations that change the state of the system were made first class (read:

not non grata, — T.U.) concepts in Prolog, rather than considering them as distasteful "side effects"...". It may be debatable whether such amendments would improve Prolog, but the spirit of Prolog would definitely be lost.

Complaints have also been made in the OO community that mergers by logic people of OO with logic are too theoretical, the terminology exploited is unfamiliar, there is too much "logic overhead" in the syntax etc. That's all true, but, in fact, nothing different has usually been intended.

In our proposal, we will not try to compromise between the convenience ideal of OO and the uniformity ideal of logic. To escape from swinging between the two, we propose a two-layer system architecture, of which we fully elaborate the internal layer, while with regard to the front-end layer, we only work out some guidelines for future design.

The internal (underlying) level will be based on logic. We will try to give explanations to OO concepts and techniques in terms of logic, and apply these to build up the computing mechanism. We will not care too much about preserving the conventional OO terminology or about traditions of syntax in this layer. Instead of that, we rather will seek to capture some general semantics of OO.

The user-visible or front-end layer should serve as an interface between the internal level and a user. Its main task would be to make the language friendly (convenient) and syntactically "OO-like" (recall that a semantics to OO will be given in the internal layer). In this layer, uniformity should be of no big concern, various shorthands in sholud be enabled etc. It should be easy to create variants for concrete applications of this layer without changing the internal layer due to the generality of the latter. Note that a similar two-layer architecture has been used in PRIZ.

In terms of implementation (not carried out in this thesis), the final execution of user-written programs could be laid on the shoulders of a Prolog interpreter. Translation from the front-end layer language into the internal layer one could be implemented through a metainterpreter written in Prolog. The internal layer language will essentially be an extension to Prolog, and could be interpreted by another metainterpreter written in Prolog. This schema is illustrated in Figure 4.1.

4.2 Trends of Structuring within the Logic Paradigm

Tendencies towards thinking more along the lines of software engineering are a relatively recent development within the logic paradigm. The techniques that are proposed, continue to be accompanied by theoretical studies. In logic programming, two directions in structuring can be observed: *modularization* (structuring of programs, i.e. grouping of clauses) and *typing* (structuring of data, i.e. grouping of individuals). Note that in OO, such a distinction between structuring directions would not work, since the OO objects and classes are a hybrid of programs and data (data-drivenness).

As examples of works in the direction of *modularization*, we mention [Mil86], [Che87], and [Day89]. A number of studies has also been carried out where modules are viewed as *worlds*, with more or less explicit parallels drawn to the possible-worlds' semantics of *modal logics*. From among the solutions reviewed in Chapter 3, Prolog/KR, Mandala, and CPU take this approach. From among proper modal enhancements of Prolog we mention two. An interesting MULTILOG proposal [KG86] has not been directly related to OO by its authors, but its possible connections are obvious. The MOLOG proposal [Far86] does not pretend to offer modularization, but instead allows a very wide understanding of modal operators. In the OO applications of the "modal logic paradigm", the *accessibility* relation

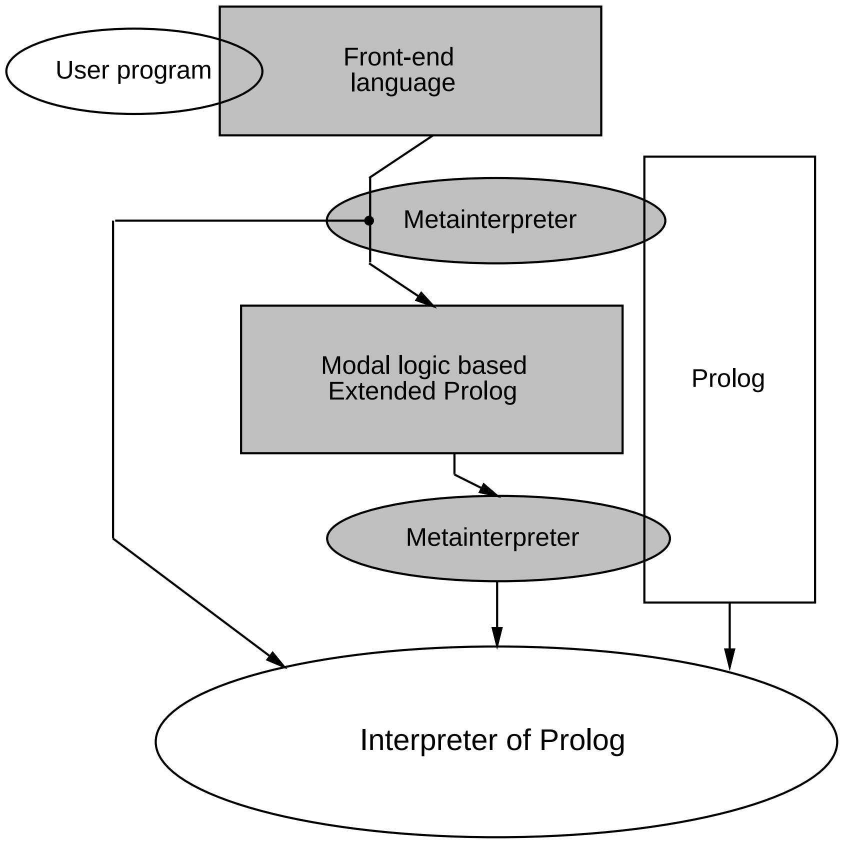

Figure 4.1: User program interpretation schema.

between worlds correlates with the *isa* relation.

Typing quite naturally leads to *many-sorted logics.* In many-sorted logics, each individual has a fixed sort, the sorts of arguments of functors and predicates are fixed, and so are the sorts of values of functors. Thus, sorts closely correspond to types. Plain many-sorted logic offers no relations between sorts. Hence, the subtype/supertype relation has no counterpart there. To provide such a counterpart, *order-sorted logics* have been developed [Wal89, Obe89, Sch89]. In order-sorted logics, a *partial ordering* is defined between sorts which basically is a subsort/supersort relation.

A good example of an order-sorted logic based language is EPOS (Extended Prolog for Order-Sorted Resolution) [HV87]. The LOGIN Prolog extension, which laid the basis for ALF, can be also viewed as relying on a variant of order-sorted logic. Its main peculiarity is that since there are no sorts in LOGIN, it is individuals that are ordered there. The ordering itself, thereby, is not programmer-declared, but automatically determined by the structure of terms denoting individuals. What "normally" would be sorts, have to be imitated via individuals in LOGIN. In a sense, the ordering in LOGIN is a hybrid between an individual/type relation and a subtype/supertype relation. In the OO context, this would correspond to a classless system where objects are ordered by means of an object/prototype relation.

The developments in the direction of typing in the logic paradigm allow OO objects to be actually modelled via individuals as such, instead of merely representing them by atoms referring to lists of clauses, like it is usually done in the modularization-prioritizing approach. The main disadvantage of proceeding from the typing instead of the modularization idea, is that dynamic states will be hard to capture. Indeed, predicates that change their interpretation in time, are much more acceptable to logic than terms that change their interpretation in time. (Cf. Vulcan (Section 3.4.10), where static objects can be represented as terms, and also comments on flexibility and rigidity in Section 4.5.)

Mainly because of the mentioned reason, in our proposal we will proceed from the modularization direction of Prolog structuring. We will also use modal logic.

Since for the logic paradigm, modules specifically are lists of clauses ("chunks of knowledge"), we decide to underline this specificity by calling them *units* within this paradigm in the rest of this thesis. This is in harmony with the terminology of [MN86].

4.3 Units. Two dimensions of evolution

In the previous section, we decided that in the design of the internal layer, we will proceed from the modularization direction of Prolog structuring. In this section, we determine what will the main building blocks, i.e. units be.

Since units are meant to create modularity, and since in OO, it is objects and classes that carry out modularity, it is natural to start from determining the logical counterpart concepts to their constituents, i.e. attributes and methods. To this end, we first must agree upon, where the main difference between these two kinds of properties is. In our opinion, the evolution of answers to this question in pure OO systems and—thereafter—in mergers with logic has been the following:

- **Attributes are hidden, methods are public** (generally!). That is an orthodox OO view, but it is more about a discipline of use of properties, rather than about what properties are (see Section 4.4 below). As such, it is right, of course.
- **Attribute values are individuals, methods are operations on them**, speaking

in logical terms. This has traditionally been so in OO systems, but it has been abandoned in most mergers with logic. The reason is that if methods are operations on attributes, then attributes necessarily are dynamic, and the most straightforward thing in generally static Prolog is predicates (through `assert`'s and `retract`'s, or in some nicer way). This leads to the next choice of the primary distinguisher.

- **Attributes are dynamic, methods are static**. This is view is characteristic to majority of the mergers with logic. Dynamics is usually achieved through `assert`'s, `retract`'s, and the like. Both attributes and methods are predicates in these mergers. Thus not only may an attribute be "multivalued" (the predicate be non-functional), but it can also be "parametric" (the predicate has some input argument places) and "conditional" (the definition clauses do have bodies). This view is pretty far from the traditional OO one, but not too far, since covers it.

The last-mentioned distinction criterion is still not fully logical, because it is not free from the "illogical" constructs `assert`, `retract`, or like. In order to get rid of them, we define *object hierarchy revisions* (shortly, revisions), as variants of the object hierarchy which emerge due to substitutionary changes being made into the total knowledge about the object hierarchy, and *revision hierarchy*, as the hierarchy determined by the temporal succession of revisions, and adopt the following distinguisher:

- **Attribute values are inherited along the revision hierarchy, methods are inherited along the object hierarchy** (we rename 'inheritance hierarchy' into 'object hierarchy' to reserve the word 'inheritance hierarchy' to become a common denominator for 'revision hierarchy' and 'object hierarchy'). This choice is justified by the following analogy. A method definition (it may be viewed as a method "value") of an object normally does not change after the object creation. But it varies from object to object. Consider the overriding mode of inheritance. Given an object that owns a definition of a method, the definition maintains its validity while moving downwards in the object hierachy, until an object is reached at that owns a different definition of the same method. Attribute values, on the contrary, vary both from object to object and from revision to revision. An attribute value of an object, assigned at a given object hierarchy revision, maintains its validity, until a revision is reached at, at which a new value is assigned to the object. We can say that the object hierarchy revision owns a value of a given attribute of a given object, if this value is being assigned to the attribute at that revision. Briefly reformulating all said, attribute values are shared between revisions, and methods are shared between objects.

The described approach allows us to treat attributes and methods in similar manner. Both will be predicates, and the only difference between them will be the way how they can be shared.

In Section 4.4, we will argue that for the purposes of the internal layer, there is no need to distinguish between classes and objects. We therefore use the term 'object' as a common denominator for 'object' and 'class' in the rest of this chapter. The term 'isa relation' will be used as a general term for both instance/class and subclass/superclass relations, and for object/prototype relation.

Taking into account the convention about the usage of the word 'object', we can state now that for the static case, it is enough that units be in a one-to-one correspondence with objects and contain the definitions (value assignments) which the respective objects own. An unit (or object) hierarchy $\mathcal{O}$ would formally be a pair $\langle O, \text{ isa } \rangle$, where O is a set of units (objects), and isa is an isa relation between them.

For the case of dynamic objects, units must be pairs ⟨object, revision⟩, and must contain the definitions (value assignments) that the respective object at the respective revision becomes owner of. Thus, a two-dimensional unit hierarchy arises, which we call the product of the two original hierarchies. Each of the dimensions—the object hierachy and the revision hierarchy—embodies one evolutionary aspect of OO. Revisions can be also thought of as time instants, and the revision hierarchy as time flow, and also called this way. To the ordinary isa relation, the relation "succeeds to" corresponds in the revision hierarchy.

The formal definition of product is the following.

Definition 4.1 (Product of hierarchies) *Assume that we are given an object hierachy $\mathcal{O} = \langle O, isa_O \rangle$, where O is a set of objects and* isa_O *is an isa relation between them, and a revision hierarchy $\mathcal{T} = \langle T, isa_T \rangle$, where T is a set of revisions and* isa_T *is an isa relation between them. Then the product of these two hierarchies is a hierarchy $\mathcal{U} = \langle U, isa_U \rangle$ of (two-dimensional) units , where the set of units U and the isa relation* isa_U *between them are defined as follows:*

$$\begin{array}{rcl} U & = & \{\langle o,t\rangle : o \in O \textit{ and } t \in T\} \\ \langle o,t\rangle \ \mathrm{isa}_U\ \langle o',t'\rangle & \textit{iff} & o\ \mathrm{isa}_O\ o' \textit{ and } t = t', \\ & & \textit{or } o = o' \textit{ and } t\ \mathrm{isa}_T\ t' \end{array}$$

The structure of a two-dimensional unit hierarchy is illustrated in Figure 4.2.

We see revisions as useful for two reasons. First, they create uniformity, since attributes and methods become analogical to each other. Second, they offer simple versioning known from software engineering and database applications. This latter argument in favor of revisions also explains why we decided to proceed from revisions of the entire object hierarchy instead of revisions of each particular object. Object revisions would not have allowed restoration of object universe revisions, because the succession of revising different objects would not have been recorded.

In his major work on time in AI [Sho88, p 19], Y. Shoham argues that two problems connected to time—the persistence problem and the frame problem—essentially coincide. The *persistence problem* lies in predicting that a piece of knowledge, valid at the present time instant, will remain valid troughout a lengthy future interval. The *frame problem* (we referred to the frame assumption in Subsection 3.4.13 already) is to assume, if a transition is made from one situation into another, and some things are claimed to change at this transition, that everything else will remain unchanged.

These two problems coincide in our logical interpretation of OO, too! Persistence means that if some property has been defined at the current object universe revision, then it normally (unless it will be redefined) maintains its current definition in the future. The frame assumption means that if the creation of the current object universe revision was not caused by a redefinition of a given revision property or object-revision property, then the last past definition of this property is still valid. It is obvious that the two formulations which we herewith made in our terminology, are reformulations of each other. The fact that inheritance along revisions provides a solution to the frame problem, was first explicitly pointed out in the MULTILOG proposal [KG86].

Property definitions (value assignments) will be represented by clauses annotated with their owners. Inter-unit *taxonomic knowledge* (i.e. the isa relations), on the contrary, will be not represented in "normal" clauses. For objects, the isa relation should be stored in a separate block of the program that essentially codes the graph of the object hierarchy, and for revisions, as we will see in Section 4.10, their ID's will implicitly determine the

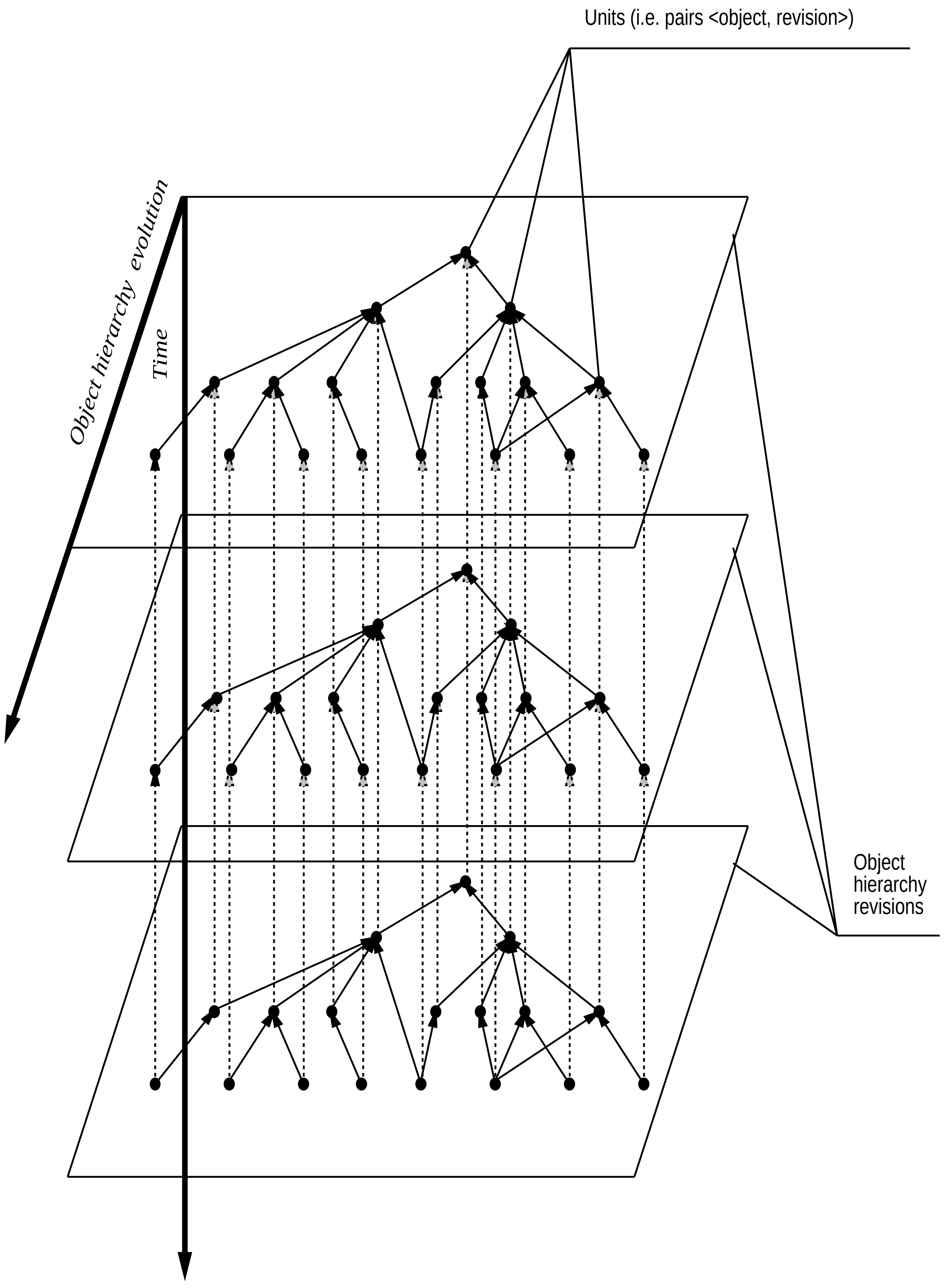
Units (i.e. pairs <object, revision>)
Object hierarchy evolution
Time
Object hierarchy revisions

Figure 4.2: Two-dimensional unit hierarchy.

isa relation. The reason for such separating of taxonomic knowledge is that we want to obtain better computational efficiency by letting unification instead of resolution to infer property bindings. The same approach has been taken in both LOGIN [AN86] and EPOS [HV87], and is particularly emphasized in the latter.

4.4 Tasks of the Layers

In order to determine the tasks of each layer, we return to the four principles of OO that we reviewed in Subsection 2.2.1. It turns out that the two first of the principles can be viewed as involving two aspects: they state what *facilities* an OO system must offer, but they also enforce a *discipline* on the usage of these facilities. We choose to follow the facilitating aspects of the OO principles in designing the logical language and its corresponding deduction mechanism of the internal layer, so to establish a broad and general fundament for our system. The restrictive functions of the OO principles will be left to be carried out by variants of language syntax of the front-end layer.

We now examine the principles in detail.

- *Data abstraction*, as mentioned in Subsection 2.2.1 already, involves *modularization* and *information hiding*. Here the division into aspects is obvious. Modularization requires our logical language to possess constructs for working with modules (units), and the deduction mechanism to be adequate w.r.t. the semantics of modules. Information hiding, on the contrary, requires no new constructs, provided that there exist means in the language for work with modules. It merely introduces restrictions on the usage of developed language (most typically, disabling direct "remote calls" to attributes, and hence forcing to use methods to access them). We consider it natural to let the syntax of the front-end language enforce such disciplines onto the user.

- *Behavior sharing* means that modules are able to share certain parts of their contents. Hence, the logics that we design should be able to handle various sharing relations between modules. What exactly must be sharable and in which manner, should be up to the front-end language syntax.

- *Evolution* means that the hierarchy of modules (i.e. the configuration and the contents of modules) is changing. We speak about *requirements evolution*, when a user refines or extends the modules' hierarchy (between program executions), and about *solving by evolution*, when the system does it (during execution). The internal layer must be able to handle the changes where the user doesn't intervene directly, i.e. time dimension must be considered. The front-end layer has to make it convenient for the user to manually change the program.

- With regards to *correctness*, we argued in Subsection 2.2.1 already that it hardly can be called a principle in the same sense as the previous ones are. However, for our two-layer architecture, it implies two guidelines. The internal layer deduction mechanism should be *sound* w.r.t. the semantics that the OO paradigm gives to the logical language. It should also be as *complete* as possible. The front-end layer should provide the *diagnostics* and *error handling* ingredients of the correctness ideal.

A summary on the aspects of the OO principles and their implications for the tasks of the layers is presented in Table 4.1. The tasks of the layers together with the general relationship of OO and logic in our merger, are also illustrated in Figure 4.3.

Principle	Internal layer	Front-end layer
	(Facilitating aspect)	(Restrictive aspect)
Data abstraction	Means for handling modules	Means for fixing hiding disciplines
Behavior sharing	Means for handling sharing relations between modules	Means for fixing sharing disciplines
Evolution	Time dimension	Means for program management
Correctness	Soundness and completeness	Diagnostics and error handling

Table 4.1: Aspects of the OO principles and their implications for the tasks of the layers.

In the internal layer, we will not distinguish between objects and classes, and talk only about objects. The reason for this is that the difference between objects and classes of class-based systems, as well as their difference of them both from objects of classless systems, lies only in the information hiding and sharing disciplines associated to each of them. (The circumstance that in class-based systems, classes serve as templates for object creation, does not matter because they also serve as templates for subclasses, and so the creation differences between objects and classes also reduce to sharing disciplines). Thus, these distinctions will appear necessary only in the front-end layer.

4.5 An Introduction to Modal Logic

The word 'logic' has two meanings: a framework for analysis, and a tool for representation and reasoning [Wal88]. The first meaning implies that logic is a branch of science; the second implies that logics (multiple!) are triples ⟨language, deduction mechanism, semantics⟩ being studied in this branch of science. The phrase '*modal logic*' likewise has two meanings. In this section we present an introduction to modal logic the branch of science, so to prepare for designing *the* particular modal logics that our internal layer relies on. We limit our study to *first order* modal logic.

Like it is with standard logics, *languages* of modal logics can be *propositional* or *predicate languages*. In the propositional case, predicates are restricted to be nullary. Terms and atomic formulae are defined as in standard logics. As a novelty, *modal operators* are introduced. Usually, there is one main modal operator $\Box$, which requires the ordinary definition of formula be augmented by the case:

If A is a formula, then $\Box A$ is a formula.

An auxiliary operator $\Diamond$, used for shorthands, is defined by the equivalence

$$\mathbf{Df}\Diamond. \quad \Diamond A \equiv \neg\Box\neg A.$$

Depending on the pragmatics of a given logic, $\Box A$ might be read as "A necessarily" (*aletic logics*), "A is known" (*logics of knowledge*), "A is believed" (*logics of belief* (*epistemic logics*)), "A is provable" (*logics of provability*) etc. In *temporal logics* (majority of them are modal) the reading of $\Box A$ most frequently is "A always" or "A always since now", but there are several more variants.

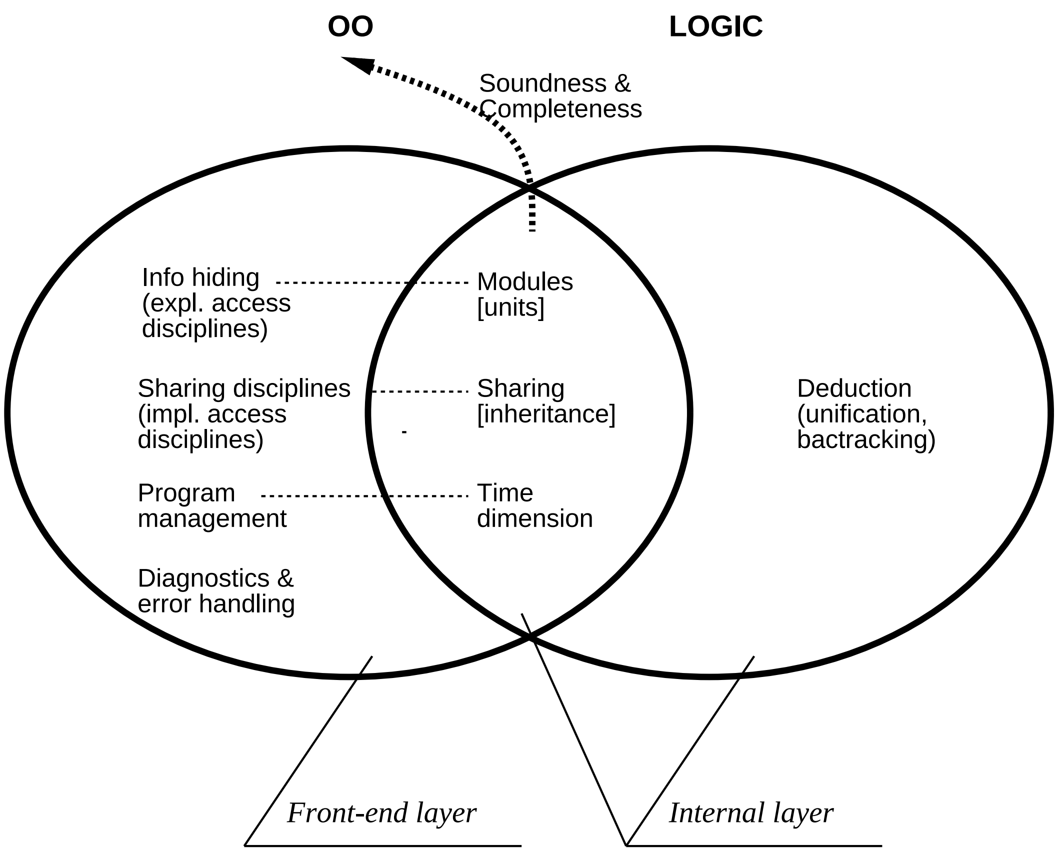
OO
LOGIC
Soundness & Completeness
Info hiding (expl. access disciplines)
Modules [units]
Sharing disciplines (impl. access disciplines)
Sharing [inheritance]
Program management
Time dimension
Diagnostics & error handling
Deduction (unification, bactracking)
Front-end layer
Internal layer

Figure 4.3: Tasks of the two layers (relationship between OO and logic).

In aletic logics and in the just-mentioned temporal logics, $\Diamond A$ also has natural readings. These are “A possibly” and “A eventually (since now)”.

Deduction mechanisms of logics can be of many different forms. A deduction mechanism can be axiomatic calculus (either Hilbert-style, or Gentzen's sequent or natural calculus), analytic tableaux, resolution calculus etc. In the following, only Hilbert-style axiomatizations and resolution calculi will be considered.

Most frequently the *Hilbert-style calculus* of a modal logic contains at least the axiom

$$\mathbf{K}.\quad \Box(A \to B) \to (\Box A \to \Box B) \quad \text{(monotonicity axiom)}$$

and the rule

$$\mathbf{N}.\quad \frac{A}{\Box A} \quad \text{(Gödel's rule, necessitation rule)},$$

and then the logic is called *normal.*

Provability in all normal logics of the following formulae is often used:

$$\mathbf{M}.\quad \Box(A\&B) \to (\Box A\&\Box B),$$

$$\mathbf{C}.\quad (\Box A\&\Box B) \to \Box(A\&B),$$

$$\mathbf{L}.\quad (\Box A \vee \Box B) \to \Box(A \vee B).$$

Typical modal axioms besides **K** are:

D. $\Box A \to \Diamond A$,

T. $\Box A \to A$,

B. $A \to \Box\Diamond A$,

4. $\Box A \to \Box\Box A$,

5. $\Diamond A \to \Box\Diamond A$.

These five axioms together with **K** and **N** yield fifteen modal logics (not 32, as it might be assumed, since these axioms are not independent). The so-called Lemmon notation for these most known normal modal logics consists in indicating the axioms' code-names. We can talk about logics KD, KT, KT4 (S4), KTB4 (=KT5, S5), KD45 etc.

There are many papers discussing how *resolution systems* for modal logics must look like, see e.g. [AM86, Cha87, Ohl88]. The main disagreements appear at two points:

- What ratio of work is to be done during the preparation phase (i.e. while transforming the negation of a given formula into Horn clauses) for deduction proper? Reformulated, which formulae must defined to be Horn clauses in a given modal logic?
- Is it resolution or unification that takes into account the modal axioms and rules?

Consider the logic S4, for example. In this logic, we have $\Box A \equiv \Box\Box A$ (it results from **T** and **4**). So at the first point of debate, we must decide whether we want to allow formulae of the form $\Box\Box A$ to be counted as literals (independently of what we would allow A to be in such cases). We might require double boxes to be removed at the transformation phase. But we might also let deduction proper to do the job. In the latter case, the transformation algorithm must augment its resulting input clause base by all clauses of the form $\Box\Box A \leftarrow \Box A$, where A's are all the literals occurring in the "normal" input clauses.

In the case when we trust the task of coping with double boxes to transformation, the formula

$$p\&(p \to \Box q) \to \Box\Box q,$$

for instance, should yield a clause set

$p \leftarrow$,
$\Box q \leftarrow p$,
$\leftarrow \Box q$.

If we want to hand the task over to deduction, the transformation will have to produce clauses

$p \leftarrow$,
$\Box q \leftarrow p$,
$\leftarrow \Box\Box q$,
$\Box\Box p \leftarrow \Box p$,
$\Box\Box q \leftarrow \Box q$.

At the second point of debate, we are faced with the choice on how we are going to deduce $p \leftarrow r$ from $p \leftarrow q$ and $\Box q \leftarrow r$, for instance. This deduction obviously requires the axiom

T to be used somehow. If we want let resolution do such deductions, our input clause set must again contain additional clauses, this time of the form $A \leftarrow \Box A$. And again it is the task of the transformation phase to insert all such auxiliary clauses. The deduction tree will then be the following:

$$\dfrac{\dfrac{p \leftarrow q \qquad q \leftarrow \Box q^{\,\mathbf{K}}}{p \leftarrow \Box q} \qquad \Box q \leftarrow r}{p \leftarrow r.}$$

Another alternative is to give the "awareness" about **T** to unification, just setting that a negative literal A and a positive literal $\Box A$ always unify. Then the deduction tree simply is:

$$\dfrac{p \leftarrow q \qquad \Box q \leftarrow r}{p \leftarrow r.}$$

The choices that particular authors make, are guided by their personal tastes. From the efficiency standpoint of implementations, however, it seems in most cases optimal to let transformation remove as much "modal load" as possible, and to put unification to cope with the remainder.

The canonical *semantics* to normal modal logics are formulated in terms of *Kripke models.* A *Kripke model* is a quadruple $\langle W, R, D, I\rangle$, where:

- W is a non-empty set whose elements are called *possible worlds*;
- R is a binary relation on W called *accessibility relation*;
- D is a non-empty set called *domain*;
- the *interpretation* I gives a meaning to each individual symbol, function symbol and predicate symbol at each world. Traditionally, individual symbols and function symbols are required to be *rigid*, i.e. their interpretations cannot change from world to world, and only predicate symbols are *flexible*, i.e. they can have different meanings in different worlds. Recently, semantics with flexible function symbols have also been studied.

The interpretation I is extended to give meanings to all expressions of the language in the following way:

- $I(w, f(t_1, \ldots, t_n)) = I(w, f)(I(w, t_1), \ldots, I(w, t_n))$;
- $I(w, p(t_1, \ldots, t_n)) = I(w, p)(I(w, t_1), \ldots, I(w, t_n))$;
- $I(w, \neg A) = \mathbf{true}$ iff $I(w, A) = \mathbf{false}$;
- $I(w, A \& B) = \mathbf{true}$ iff $I(w, A) = I(w, B) = \mathbf{true}$;
- $I(w, A \to B) = \mathbf{true}$ iff $I(w, A) = \mathbf{false}$ or $I(w, B) = \mathbf{true}$;
- $I(w, \forall x A) = \mathbf{true}$ iff for all $d \in D$, $I(w, A[\frac{x}{d}]) = \mathbf{true}$;
- $I(w, \Box A) = \mathbf{true}$ iff for all $w' \in W$, such that wRw', $I(w', A) = \mathbf{true}$.

Each logic lays certain restrictions on the accessibility relation, thus restricting the class of models the logic admits. A formula is said to be *satisfied* by a model if it is true at all the worlds of this model. A formula is *satisfiable* if there exists a model in a given model class which satisfies it. A formula is *valid* if each model of the given model class satisfies it.

Restrictions on R resulting from the most common axioms of normal modal logics, are the following:

D.	seriality	: for each $w \in W$ there exist $w' \in W$ such that wRw',
T.	reflexivity	: for each $w \in W$, wRw,
B.	symmetry	: for each $w, w' \in W$ if wRw' then $w'Rw$,
4.	transitivity	: for each $w, w', w'' \in W$ if wRw' and $w'Rw''$ then wRw'',
5.	euclidity	: for each $w, w', w'' \in W$ if wRw' and wRw'' then $w'Rw''$.

The considered axiomatizations of 15 normal modal logics are complete w.r.t. the corre-ponding Kripke semantics in the propositional case. In the predicate case, some of them turn out incomplete.

A classic introduction into modal logic is the book by B. F. Chellas [Che80].

If there is more than one modal operator in the language of a given modal logic, it is called *multimodal.* The most systematic style of denoting modal operators in multimodal logics requires that there be a set Σ_0 of *(simple) parameters* such that for each $a \in \Sigma_0$, we have a modal operator $[a]$. The writing $\langle a \rangle A$ is a shorthand for $\neg[a]\neg A$. The theory of normal multimodal logics has been developed in [Cat88]. A Hilbert-type axiomatization of a normal multimodal logic consists of the axiomatizations of a number of normal unimodal logics plus of several (maybe none) axioms of the form

$$\mathbf{G}^{\alpha,\beta,\gamma,\delta}.\quad \langle\alpha\rangle[\beta]A \to [\gamma]\langle\delta\rangle A \quad (\alpha,\beta,\gamma,\delta\text{-incestuality axiom}),$$

where $\alpha, \beta, \gamma, \delta$ are *abstract parameters.*

The set Σ of parameters is defined as follows:

1. λ is a abstract parameter.
2. If α and β are abstract parameters, then $\alpha \cup \beta$ and $\alpha;\beta$ are abstract parameters.

.

The corresponding *abstract modal operators* are understood as follows:

$$\begin{aligned} [\lambda]A &\equiv A, \\ [\alpha;\beta]A &\equiv [\alpha][\beta]A, \\ [\alpha \cup \beta]A &\equiv [\alpha]A\&[\beta]A. \end{aligned}$$

Kripke semantics can be easily generalized for the purpose of multimodal logics. Instead of having one accessibility relation R, a model now involves a set $\mathcal{R} = \{R_a : a \in \Sigma_0\}$ of them, with one per each atomic parameter. The model class for a given logic is determined by restrictions on individual accessibility relations enforced by the corresponding unimodal logics, plus by additional restrictions on the whole family of accessibility relations, that result from the incestuality axioms.

Resolution systems for multimodal logics are thoroughly discussed in [Ohl90]. The MOLOG and MULTILOG Prolog developments that we referred to in Section 4.2 already, are also both multimodal.

The next four sections (Sections 4.6–4.9) work out and study the concrete modal logics for the internal layer. The first three of them concentrate on the one-dimensional case, gradually expanding the variety of inheritance modes, while the fourth generalizes the results of the first three for the two-dimensional case.

In all these sections, given an isa relation on the set U of units, isa^* will stand for its reflexive-transitive closure, which is formally defined by the following inductive schema:

1. for any $u \in U$, $u \ \text{isa}^* \ u$;
2. for any $u, u', u'' \in U$, if u isa u' and $u' \ \text{isa}^* \ u''$ then $u \ \text{isa}^* \ u''$.

It is easy to see that for the isa relation isa_U in the product $\mathcal{U} = \langle U, \text{isa}_U \rangle$ of two hierarchies $\mathcal{O} = \langle O, \text{isa}_O \rangle$ and $\mathcal{T} = \langle T, \text{isa}_T \rangle$,

$$\langle o, t \rangle \ \text{isa}_U^* \ \langle o', t' \rangle \text{ iff } o \ \text{isa}_O^* \ o' \text{ and } t \ \text{isa}_T^* \ t'.$$

4.6 $\text{MU}_\mathcal{U}$ = Modal Units

The logic $\text{MU}_\mathcal{U}$ that this section is devoted to, is meant for reasoning about (static) unit hierarchies where "everything is inheritable".

Given a unit hierarchy $\mathcal{U} = \langle U, \text{ isa } \rangle$, $\text{MU}_\mathcal{U}$ is a normal multimodal logic with U as the set of parameters. The formula $[u]A$ is read as "A inheritably in u".

Subsection 4.6.1 describes the design motifs of $\text{MU}_\mathcal{U}$. The three following subsections formally present the axiomatics, the Kripke semantics, and the resolution calculus for the propositional fragment of $\text{MU}_\mathcal{U}$, and prove that each of these three is sound and complete w.r.t. two others. Subsection 4.6.5 comments on the full (predicate) $\text{MU}_\mathcal{U}$. In Subsection 4.6.6 explains some principal obstacles to straightforward improvements of $\text{MU}_\mathcal{U}$.

4.6.1 Motivation

L. Monteiro and A. Porto [MP90] distinguish between two orthogonal dimensions in modes of inheritance. Along the first dimension, inheritance can be *cumulative* (they call it extension mode) or *overriding*. This distinction is not new (cf. Section 3.6). Along the second, novel dimension, inheritance can be *semantic* or *syntactic*.

Suppose that u isa v. Semantic (also termed *relational* or *predicate*) inheritance by u essentially means that the literals that hold in v, are available as facts to u, i.e. that u is allowed to use the semantics of the contents of v for resolutions. Syntactic (also termed *definitional* or *clause*) inheritance by u means that the clauses that v owns, are available to u, i.e. that u is allowed to use the syntax of the contents of v for resolutions. Or, reformulated, if u inherits in the semantic mode, then in order to find the predicate interpretations in u, first the predicate interpretations are computed separately for the clause sets of u and its parents are computed separately, and then these interpretations are composed. If u inherits in the syntactic mode, then in order to find the predicate interpretations in u, first the clause sets that u and its parents own, are composed, and then the predicate interpretations for the composed clause set are computed. One could say that the difference between the semantic and syntactic modes lies in the succession of interpreting and composition.

In the light of OO, semantic inheritance roughly corresponds to the delegation of classless systems, and syntactic inheritance resembles the "inheritance" that works along the instance/class links in class-based systems. Indeed, if an object O of a classless system is supposed to perform a method, it delegates the job to the ancestor object O' that owns the corresponding definition. The state of O' may get affected through this. In class-based systems, on the contrary, an object O which is supposed to perform a method, does the job itself, and only "borrows" the relevant definition from its class C. The state of C never gets affected by this.

One more way to put the difference between semantic and syntactic inheritance in the OO terms, would be to say that in the syntactic mode, code of some owner is passed to the inheriter and executed by the inheriter, whilst in the semantic mode, code is executed by its owner, and only results are passed to the inheriter.

Altogether, the two dimensions yield four modes:

	Extension	Overriding
Predicate	PE	PO
Clause	CE	CO

The following example from [MP90] demonstrates that all the four modes really differ. Consider two units:

$$\begin{array}{|l|}\hline u: \\ \hline \\ q(1) \\ \hline \end{array} \quad \text{isa} \quad \begin{array}{|l|}\hline v: \\ \hline p(x) \leftarrow q(x) \\ q(2) \\ \hline \end{array}$$

In the unit u, the values for the arguments of the predicates in u are as follows:

	p	q
PE	2	1,2
CE	1,2	1,2
PO	2	1
CO	1	1

We will incorporate overriding neither into MU$_{\mathcal{U}}$ nor into the further logics in this thesis, and consider it only at the pragmatical level (see Subsection 4.10.1)—allowing to enforce it via cuts. But we will now give the motivation for MU$_{\mathcal{U}}$, demonstrating how the extension (cumulation) cases of the above example can be translated into modal logic in a uniform (and we dare also say, convenient) way.

The full computation process for PE and CE by u in the example is the following:

- Semantic inheritance:

$$\begin{array}{|l|l|}\hline u: & v: \\ \hline & p(x) \leftarrow q(x) \\ q(1) & q(2) \\ \hline \end{array} \quad \overset{interpreting}{\longmapsto} \quad \begin{array}{|l|l|}\hline u: & v: \\ \hline & p(2) \\ q(1) & q(2) \\ \hline \end{array} \quad \overset{composition}{\longmapsto} \quad \begin{array}{|l|}\hline u: \\ \hline p(2) \\ q(1); q(2) \\ \hline \end{array}$$

- Syntactic inheritance:

$$\begin{array}{|c|l|}\hline u: & v: \\ \hline & p(x)\leftarrow q(x) \\ q(1) & q(2) \\ \hline \end{array} \xmapsto{\textit{composition}} \begin{array}{|l|}\hline u: \\ \hline p(x)\leftarrow q(x) \\ q(1);q(2) \\ \hline \end{array} \xmapsto{\textit{interpreting}} \begin{array}{|l|}\hline u: \\ \hline p(1);p(2) \\ q(1);q(2) \\ \hline \end{array}$$

It can be easily seen that the modes differ in how the clause $p(x) \leftarrow q(x)$ which is owned by v, is used in resolutions. Independently of the mode, it can be used to resolve goals of the form $p(t)$ of u. But the unit that will be appointed to demonstrate the resolvent $q(t)$, is mode-dependent. In the semantic mode, $q(t)$ will be set up as a goal specifically for v, since interpreting will be carried out in each unit separately, and since the clause $p(x) \leftarrow q(x)$ that it must be interpreted together, is owned by v. In the syntactic mode, $q(t)$ can be viewed to become a goal for u, since the predicate interpretations for u will be computed on the basis of the composition of the clause sets owned by u and v, and in such a computation, the clause $p(x) \leftarrow q(x)$ can interact with the interpretations of $q(t)$ in both v and u.

Modal logic enables to mark the two different usages of $p(x) \leftarrow q(x)$ more explicitly.

We choose to treat units as parameters, with the initial intention that $[u]A$ should mean "A in u". But since at the present stage we assume that everything holding in some unit, can be inherited by all of its descendants, the exact meaning of $[u]A$ has to be stronger. Indeed,we can safely say $[u]A$ only if it is sure that A holds in u as well as in all its descendants. Thus, the correct reading of $[u]A$ is, "A inheritably in u".

The two steps in the procedure of interpreting under inheritance predict us two axioms:

$$\mathbf{K}_u^v. \quad [v]A \to [u]A \quad \text{if} \quad u \text{ isa } v,$$

and

$$\mathbf{K}_u. \quad [u](A \to B) \to ([u]A \to [u]B).$$

The first axiom corresponds to composition (it "copies" clauses), and the second corresponds to starting interpreting proper. For the second, the connection is more eyestriking from its following equivalent formulation:

$$[u]A\&[u](A \to B) \to [u]B.$$

Recalling that in semantic inheritance, composition takes place after interpreting proper, we see that if a clause $p \leftarrow q$ owned by v, is desired to be inheritable semantically by u, it should be translated into modal logic as:

$$[v]p \leftarrow [v]q,$$

since then only the results of interpretation proper in v, i.e. the derived facts of v, will be subject to composition. Indeed, from this translation we can obtain:

$$\frac{[u]p \leftarrow [v]p \; ^{\mathbf{K}_u^v} \qquad [v]p \leftarrow [v]q}{[u]p \leftarrow [v]q}$$

but we cannot obtain $[u]p \leftarrow [u]q$. (It could be said that the execution of the code for p is laid on the shoulders of the code's owner (note $[v]q$ in the antecedent of the conclusion).)

In syntactic inheritance by u, composition is made before interpreting proper, and the translation for same clause $p \leftarrow q$ of v would be

$$[v](p \leftarrow q),$$

since in this case, it is original input clauses that will be composed. From this translation we obtain:

$$\dfrac{\dfrac{[v](p \leftarrow q)}{[u](p \leftarrow q)\ \mathbf{K}^v_u}}{[u]p \leftarrow [u]q\ \mathbf{K}_u}$$

(It could be said that the execution of the code for p is made the task of the inheriter (note $[u]q$ in the antecedent of the conclusion).) We can also further obtain $[u]p \leftarrow [v]q$ if needed:

$$\dfrac{[u]p \leftarrow [u]q \qquad [u]q \leftarrow [v]q\ \mathbf{K}^v_u}{[u]p \leftarrow [v]q}$$

(This corresponds to the case when v owns the code of q.)

Consider how the proposed way of translation works in the example (the style of below proofs is rather informal, using side by side the classical resolution rule and some fragment of the axiomatical calculus of $\mathrm{MU}_{\mathcal{U}}$):

- Semantic inheritance:

$$\dfrac{\dfrac{\dfrac{\leftarrow [u]p(y) \qquad [u]p(y) \leftarrow [v]p(y)\ \mathbf{K}^v_u}{\leftarrow [v]p(y)} \qquad [v]p(x) \leftarrow [v]q(x)}{\leftarrow [v]q(x)} \qquad [v]q(2)}{\leftarrow \qquad y = 2}$$

- Syntactic inheritance:

$$\dfrac{\dfrac{\leftarrow [u]p(y) \qquad \dfrac{\dfrac{[v](p(x) \leftarrow q(x))}{[u](p(x) \leftarrow q(x))\ \mathbf{K}^v_u}}{[u]p(x) \leftarrow [u]q(x)\ \mathbf{K}_u}}{\leftarrow [u]q(y)} \qquad [u]q(1)}{\leftarrow \qquad y = 1}$$

$$\dfrac{\dfrac{\dfrac{\leftarrow [u]p(y) \qquad \dfrac{\dfrac{[v](p(x) \leftarrow q(x))}{[u](p(x) \leftarrow q(x))\ \mathbf{K}^v_u}}{[u]p(x) \leftarrow [u]q(x)\ \mathbf{K}_u}}{\leftarrow [u]q(y)} \qquad [u]q(y) \leftarrow [v]q(y)\ \mathbf{K}^v_u}{\leftarrow [v]q(y)} \qquad [v]q(2)}{\leftarrow \qquad y = 2}$$

Below, we will write semantically inheritable clauses in the form $[v](p \leftarrow [v]q)$. Its equivalence to the form $[v]p \leftarrow [v]q$ will be proven in the next subsection. In order to prove this equivalence, we will use an axiom

$$\mathbf{4}_{u,z}.\quad [z]A \rightarrow [u][z]A,$$

which says that if A inheritably in z then it holds inheritably in u that A inheritably in z. In other words, we assume that every unit has an access to see what holds in other units.

The purpose of the new writing for semantically inheritable clauses is to reach at a uniform form for all acceptable clauses: in each clause there will be a modality over the whole clause (the owner of the clause), and an optional modality over each literal in the body of the clause.

Consider the clause $[v](p \leftarrow [v]q, r)$. Together with a goal statement $\leftarrow [u]p$, it will give:

$$\dfrac{\leftarrow [u]p \qquad \dfrac{\dfrac{\dfrac{[v](p \leftarrow [v]q, r)}{[u](p \leftarrow [v]q, r)}\ \mathbf{K}_u^v}{[u]p \leftarrow [u][v]q, [u]r}\ \mathbf{K}_u,\mathbf{C}_u}{[u]p \leftarrow [v]q, [u]r}\ \mathbf{4}_{u,v}}{\leftarrow [v]q, [u]r}$$

The lastly considered clause produces mixed inheritance which is neither semantic nor syntactic. This is an example of a case where it is not possible to annotate pairs ⟨unit, predicate⟩, and goals must be annotated instead [MP90]. Annotating at goals, the "traditional" writing for this clause would be:

$$v: \quad p \leftarrow q, \mathrm{self}: r.$$

Besides coping with inheritance, i.e. with implicit access by unit to other units, MU$_{\mathcal{U}}$ also will allow explicit access which in OO is termed as message sending. No special language syntax is needed for this. To illustrate this, we slightly expand our last example, considering now the clause $[v](p \leftarrow q, [v]r, [z]s)$:

$$\dfrac{\leftarrow [u]p \qquad \dfrac{\dfrac{\dfrac{[v](p \leftarrow [v]q, r, [z]s)}{[u](p \leftarrow [v]q, r, [z]s)}\ \mathbf{K}_u^v}{[u]p \leftarrow [u][v]q, [u]r, [u][z]s}\ \mathbf{K}_u,\mathbf{C}_u}{[u]p \leftarrow [v]q, [u]r, [z]s}\ \mathbf{4}_{u,v},\mathbf{4}_{z,v}}{\leftarrow [v]q, [u]r, [z]s}$$

The traditional writing for this clause would be:

$$v: \quad p \leftarrow q, \mathrm{self}: r, z : s.$$

4.6.2 Axiomatics

The *Hilbert-style calculus* of MU$_{\mathcal{U}}$ consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Hilbert-style calculus.**
- **Intra-unit rules and axioms.** For each unit $u \in U$, these together with the previous group axiomatize the corresponding unimodal sublogic of MU$_{\mathcal{U}}$. $\Box$ stands

for $[u]$, where u is an arbitrary unit from U.

$$\begin{aligned}
&\mathbf{N}. && \frac{A}{\Box A}\,,\\
&\mathbf{K}. && \Box(A \to B) \to (\Box A \to \Box B),\\
&\mathbf{D}. && \Box A \to \Diamond A,\\
&\mathbf{4}. && \Box A \to \Box\Box A,\\
&\mathbf{5}. && \Diamond A \to \Box\Diamond A.
\end{aligned}$$

By $\mathbf{N}_u$, $\mathbf{K}_u$, $\mathbf{D}_u$, $\mathbf{4}_u$, and $\mathbf{5}_u$ we will denote **N**, **K**, **D**, **4**, and **5** for u, respectively.

- **Inter-unit axioms.** These determine the relationships between different modalities. For each $u, v, z \in U$ we have:

$$\begin{aligned}
&\mathbf{K}_u^v. && [v]A \to [u]A \quad \text{if} \quad u \text{ isa } v,\\
&\mathbf{4}_{u,z}. && [z]A \to [u][z]A,\\
&\mathbf{5}_{u,z}. && \langle z\rangle A \to [u]\langle z\rangle A.
\end{aligned}$$

All the three are incestuality axioms. Indeed,

$$\begin{aligned}
\mathbf{K}_u^v = \mathbf{G}^{\lambda,v,u,\lambda}: &\quad \langle\lambda\rangle[v]A \to [u]\langle\lambda\rangle A,\\
\mathbf{4}_{u,z} = \mathbf{G}^{\lambda,z,(u;z),\lambda}: &\quad \langle\lambda\rangle[z]A \to [u;z]\langle\lambda\rangle A,\\
\mathbf{5}_{u,z} = \mathbf{G}^{\lambda,z,u,z}: &\quad \langle z\rangle[\lambda]A \to [u]\langle z\rangle A.
\end{aligned}$$

Remark. It is clear that in principle, the axioms $\mathbf{4}_u$ and $\mathbf{5}_u$, can be left out from the axiomatics of the whole $\mathrm{MU}_{\mathcal{U}}$, since they coincide with $\mathbf{4}_u^u$ and $\mathbf{5}_u^u$. But we still decided to write them down explicitly, since they belong to the axiomatics of the unimodal sublogics of $\mathrm{MU}_{\mathcal{U}}$.

The following theorem shows that the condition on u, v in $\mathbf{K}_u^v$ can be weakened.

Theorem 4.1 *For each $u, v \in U$, if u isa* v, then the formula*

$$\mathbf{K}_u^v.\quad [v]A \to [u]A$$

is provable in $\mathrm{MU}_{\mathcal{U}}$.

Proof. By induction on the definition of isa* .

1. $u = v$.

$$\cfrac{\cfrac{A \to A}{[u](A \to A)}\;\mathbf{N}_u}{[u]A \to [u]A}\;\mathbf{K}_u$$

2. u isa v' and v' isa* v.

$$\frac{[v]A \to [v']A \;\mathbf{K}_u^v \text{ (ind. assumption)} \qquad [v']A \to [u]A \;\mathbf{K}_u^v \text{ (axiom)}}{[v]A \to [u]A}$$

◁

The following theorem states that in case of nested modalities only the innermost one essentially matters.

Theorem 4.2 *Given a sequence μ of modalities and a modality* m, *the formula*

$$\mu \mathrm{m} A \equiv \mathrm{m} A$$

is provable in $\mathrm{MU}_{\mathcal{U}}$.

Proof. By induction on the length of $\mathcal{M}$.

1. length(μ)=1.

$$[z]A \to [u][z]A \ \mathbf{4}_{u,z}$$

$$\frac{[u][z]A \to \langle u\rangle[z]A \ \mathbf{D}_u \qquad \dfrac{\langle z\rangle\neg A \to [u]\langle z\rangle\neg A \ 5_{u,z}}{\langle u\rangle[z]A \to [z]A}}{[u][z]A \to [z]A}$$

$$\frac{[z]A \to [u][z]A \ \mathbf{4}_{u,z} \qquad [u][z]A \to \langle u\rangle[z]A \ \mathbf{D}_u}{[z]A \to \langle u\rangle[z]A}$$

$$\frac{\langle z\rangle\neg A \to [u]\langle z\rangle\neg A \ \mathbf{5}_{u,z}}{\langle u\rangle[z]A \to [z]A}$$

$$\langle z\rangle A \to [u]\langle z\rangle A \ \mathbf{5}_{u,z}$$

$$\frac{[u]\langle z\rangle A \to \langle u\rangle\langle z\rangle A \ D_u \qquad \dfrac{[z]\neg A \to [u][z]\neg A \ \mathbf{4}_{u,z}}{\langle u\rangle\langle z\rangle A \to \langle z\rangle A}}{[u]\langle z\rangle A \to \langle z\rangle A}$$

$$\frac{\langle z\rangle A \to [u]\langle z\rangle A \ \mathbf{5}_{u,z} \qquad [u]\langle z\rangle A \to \langle u\rangle\langle z\rangle A \ \mathbf{D}_u}{\langle z\rangle A \to \langle u\rangle\langle z\rangle A}$$

$$\frac{[z]\neg A \to [u][z]\neg A \ \mathbf{4}_{u,z}}{\langle u\rangle\langle z\rangle A \to \langle z\rangle A}$$

2. $\mu = \mathrm{m}'\mu'$.

$$\frac{\mathrm{m}'\mu'\mathrm{m}A \equiv \mu'\mathrm{m}A\ ^{\text{(ind. base)}} \qquad \mu'\mathrm{m}A \equiv \mathrm{m}A\ ^{\text{(ind. assumption)}}}{\mathrm{m}'\mu'\mathrm{m}A \equiv \mathrm{m}A}$$

◁

The following theorem was referred to in the previous subsection.

Theorem 4.3 $\vdash_{\mathrm{MU}_{\mathcal{U}}} \Box(\Box A \to B) \equiv (\Box A \to \Box B)$.

Proof.

$$\cfrac{\cfrac{\Box A \to \Box\Box A\ ^{\mathbf{4}}}{(\Box\Box A \to \Box B) \to (\Box A \to \Box B)}}{\Box(\Box A \to B) \to (\Box A \to \Box B)\ ^{\mathbf{K}}}$$

$$\cfrac{\cfrac{\Diamond\neg A \to \Box\Diamond\neg A\ ^{\mathbf{5}}}{\Diamond\Box A \to \Box A} \qquad \cfrac{\cfrac{\cfrac{\cfrac{\neg A \to (\Box A \to B)}{\Box(\neg\Box A \to (\Box A \to B))\ ^{\mathbf{N}}}}{\Box\neg\Box A \to \Box(\Box A \to B)\ ^{\mathbf{K}}} \qquad \cfrac{\cfrac{B \to (\Box A \to B)}{\Box(B \to (\Box A \to B))\ ^{\mathbf{N}}}}{\Box A \to \Box(\Box A \to B)\ ^{\mathbf{K}}}}{\Box\neg\Box A \vee \Box B \to \Box(\Box A \to B)}}{(\Diamond\Box A \to B) \to \Box(\Box A \to B)}}{(\Box A \to \Box B) \to \Box(\Box A \to B)}$$

◁

The *Gentzen-style sequential calculus* for $\mathrm{MU}_{\mathcal{U}}$ consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Gentzen-style sequential calculus.**
- **Modal rules.** We take them together into one very general rule. Γ, Δ, Λ, Π, Θ and Υ denote finite sets of formulae, m denotes a modality, $\underline{\mathrm{m}}$ and $\underline{\mathrm{m}'}$ denote finite sets of modalities, and $\underline{v}$ and $\underline{v'}$ denote finite sets of units.

  $\mathbf{m}_{u}^{\underline{v}|\underline{v'}}$.

  $$\frac{\Theta, \Gamma, \underline{\mathrm{m}}\Lambda \to \Upsilon, \Delta, \underline{\mathrm{m}'}\Pi}{\langle u\rangle\Theta, [\underline{v}]\Gamma, \underline{\mathrm{m}}\Lambda \to [u]\Upsilon, \langle\underline{v'}\rangle\Delta, \underline{\mathrm{m}'}\Pi}$$

  if u isa* $\underline{v}, \underline{v'}$, and $\|\Theta \cup \Upsilon\| \leq 1$.

Theorem 4.4 *The Hilbert-style and the Gentzen-style sequential calculi of* $\mathrm{MU}_{\mathcal{U}}$ *are equivalent.*

Proof. For each of the two calculi, we have to prove that its axioms are derivable and its rules are admissible in the other. As regards the classical propositional rules and axioms this is clear. For the modal rules and axioms, the proofs follow below.

- $\mathbf{N}_u$.

$$\dfrac{\to A}{\to [u]A \;\mathbf{m}_u^{|}}$$

- $\mathbf{K}_u$.

$$\dfrac{\dfrac{\dfrac{A, B \to A \qquad A, B \to B}{A, (A \to B) \to B}}{[u]A, [u](A \to B) \to [u]B \;\mathbf{m}_u^{u|u}}}{[u](A \to B) \to ([u]A \to [u]B)}$$

- $\mathbf{K}_u^v$. u isa v.

$$\dfrac{A \to A}{[v]A \to [u]A \;\mathbf{m}_u^{v|}}$$

- $\mathbf{4}_{u,z}$.

$$\dfrac{\dfrac{A \to A}{[z]A \to [z]A \;\mathbf{m}_z^{z|}}}{[z]A \to [u][z]p \;\mathbf{m}_u^{|}}$$

- $\mathbf{5}_{u,z}$.

$$\dfrac{\dfrac{A \to A}{\langle z\rangle A \to \langle z\rangle A \;\mathbf{m}_z^{|z}}}{\langle z\rangle A \to [u]\langle z\rangle p \;\mathbf{m}_u^{|}}$$

- $\mathbf{Df}\langle u\rangle$.

$$\dfrac{\dfrac{\dfrac{A \to A}{\to A, \neg A}}{\to [u]A, \langle u\rangle\neg A \;\mathbf{m}_u^{|u}}}{\neg\langle u\rangle\neg A \to [u]A}$$

$$\dfrac{\dfrac{\dfrac{A \to A}{\neg A, A \to}}{\langle u\rangle\neg A, [u]A \to \;\mathbf{m}_u^{|u}}}{[u]A \to \neg\langle u\rangle\neg A}$$

- $\mathbf{m}_u^{\underline{v}|\underline{v'}}$.

Case 1. $\Theta = A$, $\Upsilon = \emptyset$.

$$\begin{array}{c}
A, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2 \\ \hline
\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \neg A \\ \hline
[u](\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \neg A) \ ^{\mathbf{N}_u} \\ \hline
[u]\neg\Delta, [u]\neg[\underline{z'_1}]\Pi_1, [u]\neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [u][\underline{z_1}]\Lambda_1, [u]\langle \underline{z_2} \rangle \Lambda_2 \to [u]\neg A \ ^{\mathbf{K}_u, \mathbf{C}_u} \\ \hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]\neg A \ ^{\mathbf{4}_{u,(\underline{z_1},\underline{z'_2})}, \mathbf{5}_{u,(\underline{z'_1},\underline{z_2})}} \\ \hline
[\underline{v'}]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]\neg A \ ^{\mathbf{K}_u^{(\underline{v},\underline{v'})}} \\ \hline
\langle u \rangle A, [\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \langle \underline{v'} \rangle \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2
\end{array}$$

Case 2. $\Theta = \emptyset$, $\Upsilon = A$.

$$\begin{array}{c}
\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to A, \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2 \\ \hline
\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to A \\ \hline
[u](\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to A) \ ^{\mathbf{N}_u} \\ \hline
[u]\neg\Delta, [u]\neg[\underline{z'_1}]\Pi_1, [u]\neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [u][\underline{z_1}]\Lambda_1, [u]\langle \underline{z_2} \rangle \Lambda_2 \to [u]A \ ^{\mathbf{K}_u, \mathbf{C}_u} \\ \hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]A \ ^{\mathbf{4}_{u,(\underline{z_1},\underline{z'_2})}, \mathbf{5}_{u,(\underline{z'_1},\underline{z_2})}} \\ \hline
[\underline{v'}]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]A \ ^{\mathbf{K}_u^{(\underline{v},\underline{v'})}} \\ \hline
[\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]A, \langle \underline{v'} \rangle \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2
\end{array}$$

Case 3. $\Theta \cup \Upsilon = \emptyset$, $\Gamma = A \cup \Gamma^*$.

$$\begin{array}{c}
\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2 \\ \hline
\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma^*, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \neg A \\ \hline
[u](\neg\Delta, \neg[\bar{z}'_1]\Pi_1, \neg\langle \bar{z}'_2 \rangle \Pi_2, \Gamma^*, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \neg A) \ ^{\mathbf{N}_u} \\ \hline
[u]\neg\Delta, [u]\neg[\underline{z'_1}]\Pi_1, [u]\neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma^*, [u][\underline{z_1}]\Lambda_1, [u]\langle \underline{z_2} \rangle \Lambda_2 \to [u]\neg A \ ^{\mathbf{K}_u, \mathbf{C}_u} \\ \hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma^*, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]\neg A \ ^{\mathbf{4}_{u,(\underline{z_1},\underline{z'_2})}, \mathbf{5}_{u,(\underline{z'_1},\underline{z_2})}} \\ \hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma^*, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \langle u \rangle \neg A \ ^{\mathbf{D}_u} \\ \hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \\ \hline
[\underline{v'}]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \ ^{\mathbf{K}_u^{(\underline{v},\underline{v'})}} \\ \hline
[\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \langle \underline{v'} \rangle \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2
\end{array}$$

Case 4. $\Theta \cup \Upsilon = \emptyset$, $\Delta = A \cup \Delta^*$.

$$\begin{array}{c}
\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2 \\
\hline
\neg\Delta^*, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to A \\
\hline
[u](\neg\Delta^*, \neg[\bar{z}'_1]\Pi_1, \neg\langle \bar{z}'_2 \rangle \Pi_2, \Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to A) \ ^{\mathbf{N}_u} \\
\hline
[u]\neg\Delta^*, [u]\neg[\underline{z'_1}]\Pi_1, [u]\neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [u][\underline{z_1}]\Lambda_1, [u]\langle \underline{z_2} \rangle \Lambda_2 \to [u]A \ ^{\mathbf{K}_u, \mathbf{C}_u} \\
\hline
[u]\neg\Delta^*, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to [u]A \ ^{\mathbf{4}_{u,(\underline{z_1},\underline{z'_2})}, \mathbf{5}_{u,(\underline{z'_1},\underline{z_2})}} \\
\hline
[u]\neg\Delta^*, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \langle u \rangle A \ ^{\mathbf{D}_u} \\
\hline
[u]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [u]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \\
\hline
[\underline{v'}]\neg\Delta, \neg[\underline{z'_1}]\Pi_1, \neg\langle \underline{z'_2} \rangle \Pi_2, [\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \ \mathbf{K}_u^{(\underline{v},\underline{v'})} \\
\hline
[\underline{v}]\Gamma, [\underline{z_1}]\Lambda_1, \langle \underline{z_2} \rangle \Lambda_2 \to \langle \underline{v'} \rangle \Delta, [\underline{z'_1}]\Pi_1, \langle \underline{z'_2} \rangle \Pi_2
\end{array}$$

Case 5. $\Theta \cup \Gamma \cup \Upsilon \cup \Delta = \emptyset$. This case is trivial, since the premise and the conclusion coincide.

$\lhd$

For the presented Gentzen-style calculus, cut elimination does not hold. An example is the formula

$$\begin{array}{ccccccc}
\Box(& \Box A & \& & B) \to & \Box(A & \& & \Box B) \\
1 & 2 & & & 3 & & 4
\end{array}$$

The informal explanation for this is a clash of two contradicting pairs of requirements is the following. The axioms for the hypothetical cut-free proof can only be of the forms $A, \Lambda \to A, \Pi$ and $B, \Lambda \to B, \Pi$ (there must be at least one axiom of each form). While hanging box 2 onto the antecedent A, some successor of the succedent A becomes boxed. Due to the subformula property this successor can only be $A \& \Box B$, so the boxes 2 and 3 must be introduced in the same application of the modal rule. Symmetrically, boxes 1 and 4 must be introduced in the same application of the modal rule. But, due to the structure of the formula, it is clear that box 2 must be introduced before box 1, and box 4 must be introduced before box 3. Contradiction.

In some cases, reductions can help avoid cuts (cf. the classical sequential predicate calculus where cuts can be eliminated, but not reductions):

$$\begin{array}{c}
A, \Box A \to A \qquad\qquad A, \Box A \to \Box A \\
\hline
A, \Box A \to A \& \Box A \\
\hline
\Box A, \Box A \to \Box(A \& \Box A) \\
\hline
\Box A \to \Box(A \& \Box A)
\end{array}$$

The *Ohlbach-style calculus* for $\mathrm{MU}_{\mathcal{U}}$ makes use of an auxiliary language constructs called *world paths.*

Definition 4.2 (World paths) *Assume that for each $u \in U$, we have a countable set $F_u = \{f_{u1}, f_{u2}, \ldots\}$ of symbols whose elements we call world variables of the unit u. Then:*

1. *0 is a world path.*
2. *Every world variable is a world path.*

Formulae in this extended language are labelled with world paths (labels are written as parenthetical prefixes). The calculus contains the following rules and axioms:

- **Labelled versions of the rules and axioms of the classical propositional Gentzen-style sequential calculus.** In the axiom, the axiom formula must have same label in the antecedent and in the consequent. In the cut, the cut formula must have same label in both premises. In all the other rules, the major and the minor formulae must have same label.
- **Modal rules.** They are four in number. g denotes a world path, $\underline{h}$ and $\underline{h'}$ denote finite sets of world paths, f denotes a world variable, and unit(f) is the unit of f.

  $[v]\!\rightarrow$.

  $$\frac{(f)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)[v]A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

  where unit(f) isa* v.

  $\rightarrow\![u]$.

  $$\frac{(\underline{h})\Lambda \rightarrow (f)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)[u]A, (\underline{h'})\Pi}$$

  where unit$(f) = u$ and f does not occur anywhere in the conclusion sequent and lower in the proof.

  $\langle u\rangle\!\rightarrow$.

  $$\frac{(f)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)\langle u\rangle A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

  where unit$(f) = u$ and f does not occur anywhere in the conclusion sequent and lower in the proof.

  $\rightarrow\!\langle v\rangle$.

  $$\frac{(\underline{h})\Lambda \rightarrow (f)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)\langle v\rangle A, (\underline{h'})\Pi}$$

  where unit(f) isa* v.

Theorem 4.5 *The Gentzen-style sequential and the Ohlbach-style calculi of* $\mathrm{MU}_{\mathcal{U}}$ *are equivalent in the sense that a sequent* $\Lambda \rightarrow \Pi$ *is provable in Gentzen style iff the sequent* $(0)\Lambda \rightarrow (0)\Pi$ *is provable in Ohlbach style.*

Proof. If-only part. We indicate a bottom-up transformation procedure of Gentzen-style proofs into Ohlbach-style proofs. At each step, we guarantee that that the non-modalized formulae of the premise of the rule application under transformation get equal labels in the correponding sequent of the resulting proof.

We consider only the modal rule $\mathbf{m}_u^{\underline{v}|\underline{v'}}$, the rest are trivial. In the first two of the three cases below, it is assumed that the world variable f is of the unit u and occurs nowhere in the already built fragment of the resulting proof. $\underline{h}$, $\underline{h'}$, $\underline{k}$ and $\underline{k'}$ denote world paths.

Case 1. $\Theta = A$, $\Upsilon = \emptyset$.

$$\dfrac{\dfrac{(f)A, (f)\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (f)\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}{(\underline{q})[\underline{v}]\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (f)A, (\underline{q'})\langle\underline{v'}\rangle\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}\ {}^{[\underline{v}]\mapsto, \to\langle\underline{v'}\rangle}}{(g)\langle u\rangle A, (\underline{q})[\underline{v}]\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (\underline{q'})\langle\underline{v'}\rangle\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}\ {}^{\langle u\rangle\mapsto}$$

Case 2. $\Theta = \emptyset$, $\Upsilon = A$.

$$\dfrac{\dfrac{(f)\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (f)A, (f)\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}{(\underline{h})[\underline{v}]\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (f)A, (\underline{h'})\langle\underline{v'}\rangle\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}\ {}^{[\underline{v}]\mapsto, \to\langle\underline{v'}\rangle}}{(\underline{h})[\underline{v}]\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (g)[u]A, (\underline{h'})\langle\underline{v'}\rangle\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}\ {}^{\to[u]}$$

Case 3. $\Theta \cup \Upsilon = \emptyset$.

$$\dfrac{(f)\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (f)\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}{(\underline{h})[\underline{v}]\Gamma, (\underline{k})\underline{\mathrm{m}}\Lambda \to (\underline{h'})\langle\underline{v'}\rangle\Delta, (\underline{k'})\underline{\mathrm{m}'}\Pi}\ {}^{[\underline{v}]\mapsto, \to\langle\underline{v'}\rangle}$$

If part. For each sequent Σ of the Ohlbach-style language we define its translation $\Sigma\sharp$ into the usual label-free language. For all axioms and rules of the Ohlbach-style calculus we then show that their translations are provable/admissible in the Hilbert-style calculus which, as we know from Theorem 4.4 already, is equivalent to the Gentzen-style sequential calculus.

Let Σ_g denote the label-free sequent obtained from Σ by filtrating out its member formulae labelled by g. Suppose F^* is the set of all world variables occurring in Σ. Then

$$\Sigma\sharp = \Sigma_0 \ \vee \bigvee_{f \in F^*} [\mathrm{unit}(f)]\Sigma_f.$$

- The classical axiom.
  Case 1. $g = 0$.

$$(A, Q \to A, Q') \vee R$$

  Case 2. $g \neq 0$.

$$\dfrac{(A, Q \to A, Q') \vee R}{[z](A, Q \to A, Q') \vee R}\ {}^{\mathbf{N}_z}$$

- Classical rules. We conside two-premise rules. One-premise rules are even simpler.
  Case 1. $g = 0$.

$$\dfrac{\dfrac{S_1 \vee R \qquad S_2 \vee R}{S_1 \& S_2 \vee R} \qquad S_1 \& S_2 \to S}{S \vee R}$$

Case 2. $g \neq 0$.

$$\dfrac{\dfrac{\dfrac{[z]S_1 \vee R \qquad [z]S_2 \vee R}{[z]S_1 \& [z]S_2 \vee R}}{[z](S_1 \& S_2) \vee R} \qquad \dfrac{\dfrac{S_1 \& S_2 \to S}{[z](S_1 \& S_2 \to S)\ ^{\mathbf{N}_z}}}{[z](S_1 \& S_2) \to [z]S\ ^{\mathbf{K}_z}}}{S \vee R}$$

- $[v]{\to}$.
  Case 1. $g = 0$.

$$\dfrac{\dfrac{\dfrac{(Q \to Q') \vee [u](A, T \to T') \vee R}{(Q \to Q') \vee ([u]A \to [u](T \to T')) \vee R\ ^{\mathbf{K}_u}}}{(Q \to Q') \vee ([v]A \to [u](T \to T')) \vee R\ ^{\mathbf{K}_u^v}}}{([v]A, Q \to Q') \vee [u](T \to T') \vee R}$$

  Case 2. $g = f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[u](A, Q \to Q') \vee R}{([u]A \to [u](Q \to Q')) \vee R\ ^{\mathbf{K}_u}}}{([v]A \to [u](Q \to Q')) \vee R\ ^{\mathbf{K}_u^v}}}{[u]([v]A \to) \vee [u](Q \to Q') \vee R\ ^{\text{Th. 4.2}}}}{[u]([v]A, Q \to Q') \vee R\ ^{\mathbf{L}_u}}$$

  Case 3. $g \neq 0$, $g \neq f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[z](Q \to Q') \vee [u](A, T \to T') \vee R}{[z](Q \to Q') \vee ([u]A \to [u](T \to T')) \vee R\ ^{\mathbf{K}_u}}}{[z](Q \to Q') \vee ([v]A \to [u](T \to T')) \vee R\ ^{\mathbf{K}_u^v}}}{[z](Q \to Q') \vee [z]([v]A \to) \vee [u](T \to T') \vee R\ ^{\text{Th. 4.2}}}}{[z]([v]A, Q \to Q') \vee [u](T \to T') \vee R\ ^{\mathbf{L}_z}}$$

- ${\to}[u]$.
  Case 1. $g = 0$

$$\dfrac{(Q \to Q') \vee R \vee (\to [u]A)}{(Q \to [u]A, Q') \vee R}$$

  Case 2. $g \neq 0$.

$$\dfrac{\dfrac{[z](Q \to Q') \vee R \vee (\to [u]A)}{[z](Q \to Q') \vee R \vee [z](\to [u]A)\ ^{\text{Th. 4.2}}}}{[z](Q \to [u]A, Q') \vee R\ ^{\mathbf{L}_z}}$$

- $\langle u \rangle \to$.
  Case 1. $p = 0$

$$\begin{array}{c}
(Q \to Q') \lor R \lor (\langle u \rangle A \to) \\ \hline
(\langle u \rangle A, Q \to Q') \lor R
\end{array}$$

Case 2. $g \neq 0$.

$$\begin{array}{cl}
[z](Q \to Q') \lor R \lor (\langle u \rangle A \to) & \\ \hline
[z](Q \to Q') \lor R \lor [z](\langle u \rangle A \to) & \text{Th. 4.2} \\ \hline
[z](\langle u \rangle A, Q \to Q') \lor R & \mathbf{L}_z
\end{array}$$

- $\to \langle v \rangle$.
  Case 1. $g = 0$.

$$\begin{array}{cl}
(Q \to Q') \lor [u](T \to A, T') \lor R & \\ \hline
(Q \to Q') \lor [u](\neg A \to (T \to T')) \lor R & \\ \hline
(Q \to Q') \lor ([u]\neg A \to [u](T \to T')) \lor R & \mathbf{K}_u \\ \hline
(Q \to Q') \lor ([v]\neg A \to [u](T \to T')) \lor R & \mathbf{K}_u^v \\ \hline
(Q \to Q') \lor [u](T \to T') \lor \langle v \rangle A \lor R & \\ \hline
(Q \to \langle v \rangle A, Q') \lor [u](T \to T') \lor R &
\end{array}$$

Case 2. $g = f$.

$$\begin{array}{cl}
[u](Q \to A, Q') \lor R & \\ \hline
[u](\neg A \to (Q \to Q')) \lor R & \\ \hline
([u]\neg A \to [u](Q \to Q')) \lor R & \mathbf{K}_u \\ \hline
([v]\neg A \to [u](Q \to Q')) \lor R & \mathbf{K}_u^v \\ \hline
[u](Q \to Q') \lor \langle v \rangle A \lor R & \\ \hline
[u](Q \to Q') \lor [u]\langle v \rangle A \lor R & \text{Th. 4.2} \\ \hline
[u](Q \to \langle v \rangle A, Q') \lor R & \mathbf{L}_u
\end{array}$$

Case 3. $g \neq 0$, $g \neq f$.

$$\begin{array}{cl}
[z](Q \to Q') \lor [u](T \to A, T') \lor R & \\ \hline
[z](Q \to Q') \lor [u](\neg A \to (T \to T')) \lor R & \\ \hline
[z](Q \to Q') \lor ([u]\neg A \to [u](T \to T')) \lor R & \mathbf{K}_u \\ \hline
[z](Q \to Q') \lor ([v]\neg A \to [u](T \to T')) \lor R & \mathbf{K}_u^v \\ \hline
[z](Q \to Q') \lor [u](T \to T') \lor \langle v \rangle A \lor R & \\ \hline
[z](Q \to Q') \lor [u](T \to T') \lor [z]\langle v \rangle A \lor R & \text{Th. 4.2} \\ \hline
[z](Q \to \langle v \rangle A, Q') \lor [u](T \to T') \lor R & \mathbf{L}_z
\end{array}$$

$\triangleleft$

Theorem 4.6 *In the Ohlbach-style calculus for* $\mathrm{MU}_{\mathcal{U}}$*, cuts can be eliminated.*

Proof. The standard cut elimination procedure for the classical first-order predicate logic goes through.

$\triangleleft$

Theorem 4.7 $\mathrm{MU}_{\mathcal{U}}$ *is decidable.*

Proof. The decision procedure is the following. Given a formula, we build its reduction tree, and leave the labels of the minor formulae of applications of $[v]\!\rightarrow$ and $\rightarrow\!\langle v\rangle$ "open". More precisely, at each of such applications, we give the minor formula a label X, where X is a metavariable ("meta" in the sense that it does not belong to the language) of the unit v, and has not occurred lower. From a so-built reduction tree, a proof tree can be obtained iff the leaves can be transformed into axioms. For a given leaf sequent this is possible iff there is a formula there which is a member of both the antecedent and the consequent, and its corresponding labels satisfy one of the following conditions:

- they are both paths and they are identical,
- one of them is a path (say f), and the other is a metavariable (say X), and $\mathrm{unit}(f)$ isa* $\mathrm{unit}(X)$,
- they are both metavariables (say X and X'), and there exists $u \in U$ such that u isa* $\mathrm{unit}(X), \mathrm{unit}(X')$.

Since the construction of the reduction tree always terminates, and since each of the finite number of leafs of this tree has a finite number of pairs of antecedent and consequent formulae, and since the check for each of such pairs terminates, the whole procedure always terminates.

$\triangleleft$

4.6.3 Semantics

The restrictions on the accessibility relations in the model class (the Kripke semantics) of $\mathrm{MU}_{\mathcal{U}}$ are the following:

$\mathbf{D}_u$-M.	seriality	: for any $w \in W$ there exist $w' \in W$ such that wR_uw',
$\mathbf{K}_u^v$-M.	inclusion	: if u isa v then for any $w, w' \in W$ if wR_uw' then wR_vw',
$\mathbf{4}_{u,z}$-M.	quasi-transitivity	: for any $w, w', w'' \in W$ if wR_uw' and $w'R_zw''$ then wR_zw'',
$\mathbf{5}_{u,z}$-M.	quasi-euclidity	: for any $w, w', w'' \in W$ if wR_uw' and wR_zw'' then $w'R_zw''$.

Theorem 4.8 (Soundness) *If a formula is provable in* $\mathrm{MU}_{\mathcal{U}}$*, then it is valid in the Kripke semantics of* $\mathrm{MU}_{\mathcal{U}}$*, i.e. it is satisfied by every model meeting the restrictions* $\mathbf{D}_u$-M*,* $\mathbf{K}_u^v$-M*,* $\mathbf{4}_{u,z}$-M*, and* $\mathbf{5}_{u,z}$-M*. Moreover, none of these restrictions can be left out.*

Proof. The classical propositional rules and axioms, the rule $\mathbf{N}_u$, and the axiom $\mathbf{K}_u$ are validity preserving transitions/valid formulae in any class of Kripke models. It is therefore sufficient to prove for each axiom **Ax** from among the axioms $\mathbf{D}_u$, $\mathbf{K}_u^v$, $\mathbf{4}_{u,z}$, and $\mathbf{5}_{u,z}$, that:

1. if a model meets **Ax**-M, then **Ax** is satisfied by that model;
2. there exists a model which meets all the restrictions except for **Ax**-M, and which does not satisfy **Ax**.

- $\mathbf{D}_u$. (We skip the index u.)

$$\begin{aligned}
&\forall w \exists w'\, wRw' \\
&\equiv \forall w \exists w' [wRw' \& (\neg I(w',A) = \mathbf{true} \vee I(w',A) = \mathbf{true})] \\
&\equiv \forall w \exists w' [(wRw' \& \neg I(w',A) = \mathbf{true}) \vee (wRw' \& I(w',A) = \mathbf{true})] \\
&\equiv \forall w \exists w' [(wRw' \to \neg I(w',A) = \mathbf{true}) \to (wRw' \& I(w',A) = \mathbf{true})] \\
&\equiv \forall w [\forall w' (wRw' \to \neg I(w',A) = \mathbf{true}) \to \exists w' (wRw' \& I(w',A) = \mathbf{true})] \\
&\equiv \forall w [I(w, \Box A) = \mathbf{true} \to I(w, \Diamond A) = \mathbf{true}] \\
&\equiv \forall w\, I(w, \Box A \to \Diamond A) = \mathbf{true}
\end{aligned}$$

- $\mathbf{K}_u^v$. u isa v.

  1.

$$\begin{aligned}
&\forall w w' [wR_u w' \to wR_v w'] \\
&\to \forall w w' [wR_u w' \& \neg I(w',A) = \mathbf{true} \to wR_v w' \& \neg I(w',A) = \mathbf{true}] \\
&\equiv \forall w w' [\neg(wR_u w' \to I(w',A) = \mathbf{true}) \to \neg(wR_v w' \to I(w',A) = \mathbf{true}] \\
&\equiv \forall w w' [(wR_v w' \to I(w',A) = \mathbf{true}) \to (wR_u w' \to I(w',A) = \mathbf{true}] \\
&\to \forall w [\forall w' (wR_v w' \to I(w',A) = \mathbf{true}) \to \forall w' (wR_u w' \to I(w',A) = \mathbf{true}] \\
&\equiv \forall w [I(w, [v]A) = \mathbf{true} \to I(w, [u]A) = \mathbf{true}] \\
&\equiv \forall w\, I(w, [v]A \to [u]A) = \mathbf{true}
\end{aligned}$$

  2. Consider a model (see Figure 4.4), where

$$\begin{aligned}
W &= \{w, w_u, w_v\}, \\
R_u &= \{(w, w_u), (w_u, w_u), (w_v, w_u)\}, \\
R_v &= \{(w, w_v), (w_u, w_v), (w_v, w_v)\}, \\
I(w_u, p) &= \mathbf{false}, \\
I(w_v, p) &= \mathbf{true}.
\end{aligned}$$

  In this model

$$\neg(wR_u w_u \to wR_v R_u)$$

  and

$$I(w, [u]p \to [u]p) = \mathbf{false}.$$

  The model meets $\mathbf{D}_u$-M, $\mathbf{4}_{u,z}$-M, and $\mathbf{5}_{u,z}$-M.

- $\mathbf{4}_{u,z}$.

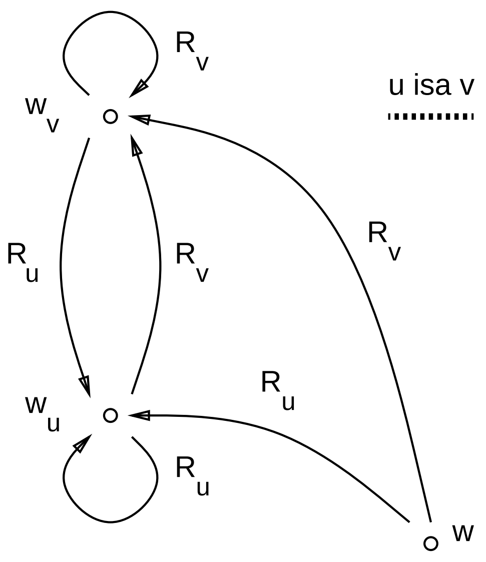

Figure 4.4: A countermodel to redundancy of $\mathbf{K}_u^v$-M.

1.

$$\begin{aligned}
&\forall ww'w''[wR_uw'\&w'R_zw'' \to wR_zw''] \\
\to\ & \forall ww'w''[wR_uw'\&w'R_zw''\&\neg I(w'',A) = \mathbf{true} \\
& \qquad \to wR_zw''\&\neg I(w'',A) = \mathbf{true}] \\
\equiv\ & \forall ww'w''[\neg(wR_uw'\&w'R_zw'' \to I(w'',A) = \mathbf{true}) \\
& \qquad \to \neg(wR_zw'' \to I(w'',A) = \mathbf{true})] \\
\equiv\ & \forall ww'w''[(wR_zw'' \to I(w'',A) = \mathbf{true}) \\
& \qquad \to (wR_uw' \to (w'R_zw'' \to I(w'',A) = \mathbf{true}))] \\
\to\ & \forall ww'[\forall w''(wR_zw'' \to I(w'',A) = \mathbf{true}) \\
& \qquad \to (wR_uw' \to \forall w''(w'R_zw'' \to I(w'',A) = \mathbf{true}))] \\
\equiv\ & \forall w[\forall w''(wR_zw'' \to I(w'',A) = \mathbf{true}) \\
& \qquad \to \forall w'(wR_uw' \to \forall w''(w'R_zw'' \to I(w'',A) = \mathbf{true}))] \\
\equiv\ & \forall w[I(w,[z]A) = \mathbf{true} \to I(w,[u][z]A) = \mathbf{true}] \\
\equiv\ & \forall w\ I(w,[z]A \to [u][z]A) = \mathbf{true}
\end{aligned}$$

2. Consider a model (see Figure 4.5), where

$$\begin{aligned}
W &= \{w, w_u, w_{z1}, w_{z2}\}, \\
R_u &= \{(w,w_u),(w_u,w_u),(w_{z1},w_u),(w_{z2},w_u)\}, \\
R_z &= \{(w,w_{z1}),(w_u,w_{z1}),(w_{z1},w_{z1}),(w_{z2},w_{z1}), \\
&\quad (w_u,w_{z2}),(w_{z1},w_{z2}),(w_{z2},w_{z2})\}, \\
I(w_{z1},p) &= \mathbf{true}, \\
I(w_{z2},p) &= \mathbf{false}.
\end{aligned}$$

In this model

$$\neg(wR_uw_u\&w_uR_zR_{z2} \to wR_zR_{z2})$$

and

$$I(w,[z]p \to [u][z]p) = \mathbf{false}.$$

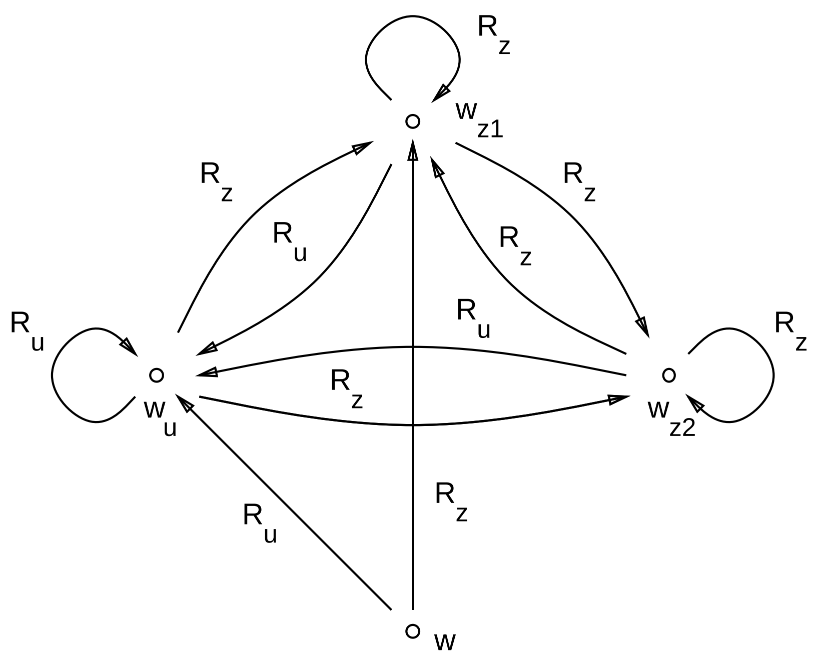

Figure 4.5: A countermodel to redundancy of $\mathbf{4}_{u,z}$-M.

The model meets $\mathbf{D}_u$-M, $\mathbf{K}^v_u$-M, and $\mathbf{5}_{u,z}$-M.

- $\mathbf{5}_{u,z}$.

1.

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall ww'w''[wR_uw'\&wR_zw'' \to w'R_zw'']} \\
\equiv & \forall ww'w''[wR_uw' \to (wR_zw'' \to w'R_zw'')] \\
\to & \forall ww'w''[wR_uw'\&I(w'',A) = \mathbf{true} \\
 & \qquad \to (wR_zw'' \to w'R_zw''\&I(w'',A) = \mathbf{true})] \\
\to & \forall ww'[\exists w''(wR_uw'\&I(w'',A) = \mathbf{true}) \\
 & \qquad \to (wR_zw'' \to \exists w''(w'R_zw''\&I(w'',A) = \mathbf{true}))] \\
\equiv & \forall w[\exists w''(wR_uw'\&I(w'',A) = \mathbf{true}) \\
 & \qquad \to \forall w'(wR_zw'' \to \exists w''(w'R_zw''\&I(w'',A) = \mathbf{true}))] \\
\equiv & \forall w[I(w,\langle z\rangle A) = \mathbf{true} \to I(w,[u]\langle z\rangle A) = \mathbf{true}] \\
\equiv & \forall w\ I(w,\langle z\rangle A \to [u]\langle z\rangle A) = \mathbf{true}
\end{array}$$

2. Consider a model (see Figure 4.6), where

$$\begin{array}{rcl}
W & = & \{w, w_u, w_{z1}, w_{z2}\}, \\
R_u & = & \{(w,w_u),(w_u,w_u),(w_{z1},w_u),(w_{z2},w_u)\}, \\
R_z & = & \{(w,w_{z1}),(w_{z1},w_{z1}), \\
 & & (w,w_{z2}),(w_u,w_{z2}),(w_{z2},w_{z2})\}, \\
I(w_{z1},p) & = & \mathbf{true}, \\
I(w_{z2},p) & = & \mathbf{false}.
\end{array}$$

In this model

$$\neg(wR_uw_u\&wR_zR_{z1} \to w_uR_zR_{z1})$$

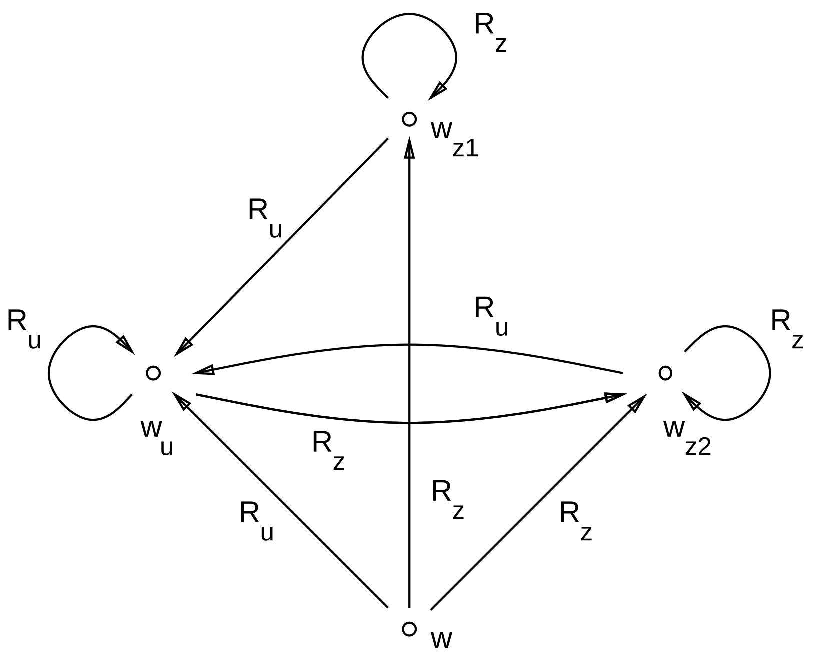

Figure 4.6: A countermodel to redundancy of $\mathbf{5}_{u,z}$-M.

and

$$I(w, \langle z\rangle p \rightarrow [u]\langle z\rangle p) = \textbf{false}.$$

The model meets $\mathbf{D}_u$-M, $\mathbf{K}_u^v$-M, and $\mathbf{4}_{u,z}$-M.

$\lhd$

Definition 4.3 (Standard models) *Assume we are given a unit hierarchy $\mathcal{U} = \langle U, \text{ isa } \rangle$. Then we say that a Kripke model $\langle W, \{R_u : u \in U\}, I\rangle$ is a standard model of* $\mathrm{MU}_{\mathcal{U}}$*, if:*

- *$W = \{0\} \cup \bigcup_{u \in U} W_u$, where W_u's are pairwise non-intersecting non-empty finite sets,*
- *and for any $w, w' \in W$ and $u \in U$,*

$$wR_uw' \textit{ iff there exists } u' \in U \textit{ such that } w' \in W'_u \textit{ and } u' \text{ isa}^* \ u.$$

Theorem 4.9 *For any $\mathcal{U}$, the standard models of* $\mathrm{MU}_{\mathcal{U}}$ *belong to the model class of* $\mathrm{MU}_{\mathcal{U}}$*.*

Proof. We have to prove that standard models meet the restrictions $\mathbf{D}_u$-M, $\mathbf{K}_u^v$-M, $\mathbf{4}_{u,z}$-M, and $\mathbf{5}_{u,z}$-M.

- $\mathbf{D}_u$-M. Choose an element $w_u \in W_u$ (this is always possible since W_u is required to be non-empty). For any $w \in W$, wR_uw_u.
- $\mathbf{K}_u^v$-M. Assume that u isa v, $w, w' \in W$, and wR_uw'. The latter is equivalent to that there exists $u' \in U$ such that $w' \in W'_u$ and u' isa* u, which together with u isa v yields u' isa* v. This in turn is equivalent to wRv'_w.

- $\mathbf{4}_{u,z}$-M and $\mathbf{5}_{u,z}$-M. For any $w, w', w'' \in W$, $w' R_z w''$ is equivalent to that there exists $z' \in U$ such that $w'' \in W_{z'}$ and z' isa* z. This in turn is equivalent to $w R_z w''$.

$\lhd$

Theorem 4.10 (Completeness) *If a formula is valid in the Kripke semantics of* $\mathrm{MU}_{\mathcal{U}}$*, i.e. if it is satisfied by every model meeting the restrictions* $\mathbf{D}_u$-M*,* $\mathbf{K}_u^v$-M*,* $\mathbf{4}_{u,z}$-M*, and* $\mathbf{5}_{u,z}$-M*, then it is provable in* $\mathrm{MU}_{\mathcal{U}}$*.*

Proof. We show that if a formula is not provable in $\mathrm{MU}_{\mathcal{U}}$, then there is a model in the Kripke semantics of $\mathrm{MU}_{\mathcal{U}}$ which does not satisfy this formula.

Assume that some formula C is not provable. This means that its reduction tree cannot be rebuilt into a proof. Choose a leaf $\Lambda \to \Pi$ of the reduction tree, which cannot be transformed into an axiom. We now can define the following standard model:

$$\begin{aligned}
W_u &= \{f : f \in F_u \text{ and } f \text{ occurs in the reduction tree,}\} \\
I(0,p) &= \begin{cases} \mathbf{true} & \text{if } \Lambda \text{ contains } (0)p, \\ \mathbf{false} & \text{if } \Pi \text{ contains } (0)p, \\ \text{arbitrary} & \text{otherwise,} \end{cases} \\
I(f,p) &= \begin{cases} \mathbf{true} & \text{if } \Lambda \text{ contains } (f)p, \\ \mathbf{true} & \text{if } \Lambda \text{ contains } (X)p \text{ where unit}(f) \text{ isa}^* \text{ unit}(X), \\ \mathbf{false} & \text{if } \Pi \text{ contains } (f)p, \\ \mathbf{false} & \text{if } \Pi \text{ contains } (X)p \text{ where unit}(f) \text{ isa}^* \text{ unit}(X), \\ \text{arbitrary} & \text{otherwise.} \end{cases}
\end{aligned}$$

Translate now each formula $(g)A$ as $I(g, A) = \mathbf{true}$, each antecedent formula $(X)A$ as $\forall u(u \text{ isa}^* \text{ unit}(X) \to (\forall f \in F_u) I(f^u, A) = \mathbf{true})$, and each consequent formula $(X)A$ as $\exists u(u \text{ isa}^* \text{ unit}(X) \to (\forall f \in F_u) I(f^u, A) = \mathbf{true})$. By induction on the height of the reduction tree, we prove that on the thread from the chosen leaf to the root, translations of the antecedent member formulae are true, and translations of the consequent member formulae are false.

For the members the leaf sequent, this holds by the definition of our model. For all the other sequents, we make use of the inductive assumption about the premise sequent(s) of the rule application that produced it. The case of classical propositional rules is simple. In the case of the modal rules we rely on the following arguments:

- $[v]\!\to$. By the inductive assumption, we have that $\forall u(u \text{ isa}^*\ v \to (\forall f \in F_u) I(f, A) = \mathbf{true})$, which due to standardness of the model is equivalent to $\forall f(g R_v f \to I(f, A) = \mathbf{true})$, i.e. to $I(g, [v]A) = \mathbf{true}$.
- $\to\![u]$. By the inductive assumption, we have that $I(f, A) = \mathbf{false}$, where f is of unit u. Due to standardness of the model, $g R_u f$. Hence, $I(g, [u]A) = \mathbf{false}$.
- $\langle u\rangle\!\to$. By the inductive assumption, we have that $I(f, A) = \mathbf{true}$, where f is of unit u. Due to standardness of the model, $g R_u f$. Hence, $I(g, \langle u\rangle A) = \mathbf{true}$.
- $\to\!\langle v\rangle$. By the inductive assumption, we have that $\neg\exists u(u \text{ isa}^*\ v \to (\forall f \in F_u) I(f, A) = \mathbf{true})$, which due to standardness of the model is equivalent to $\neg\exists f(g R_v f \to I(f, A) = \mathbf{true})$, i.e. to $I(g, \langle v\rangle A) = \mathbf{false}$.

The end formula $(0)C$ in the consequent of the root translates into $I(0, C) = \mathbf{true}$. According to what we just proved, this translation is false. Hence, C is not satisfied by the model.

$\lhd$

We now define a concept of *simple models*, where units and worlds are in a one-to-one correspondence (the models corresponding to naive intuition).

Definition 4.4 (Simple models) *Assume we are given a unit hierarchy $\mathcal{U} = \langle \mathcal{U}, \text{ isa } \rangle$. Then we say that a Kripke model $\langle W, \{R_u : u \in U\}, I\rangle$ is a simple model of* MU$_{\mathcal{U}}$*, if:*

- $W = \{0\} \cup U$,
- *and for any $w, w' \in W$ and $u \in U$,*

$$wR_uw' \textit{ iff } w' \in U \textit{ and } w' \text{ isa}^* u.$$

It is obvious that every simple model is standard model.

In the simple models, the usual definition of I at modalized formulae:

$$I(w, [u]A) = \mathbf{true} \text{ iff for all } w' \in W \text{ such that } wR_uw',\ I(w', A) = \mathbf{true},$$

becomes equivalent to:

$$I(w, [u]A) = \mathbf{true} \text{ iff for all } w' \in U \text{ such that } w' \text{ isa}^* u,\ I(w', A) = \mathbf{true},$$

and this is in concert with the intuitively clear idea that A can inheritably hold in u for w only if A holds at all the units inheriting from u.

The relationship between isa and accessibility relations in simple models is illustrated by the example in Figure 4.7.

Note that a formula that is satisfied by every simple model of MU$_{\mathcal{U}}$, need not be provable in MU$_{\mathcal{U}}$. An example is the formula

$$[u](p \vee q) \rightarrow [u]p \vee [u]q$$

for $\mathcal{U}$ where there exist no $v \in U$ such that v isa u. In simple models in such a case, for every formula A and for every $w \in W$, $I(w, [u]A) = I(u, A)$, and hence our formula is always satisfied. The following standard model, however, does not satisfy it, which means that it is not provable:

$$\begin{array}{rcl} W_u & = & \{w_{u1}, w_{u2}\} \\ I(w_{u1}, p) & = & \mathbf{true} \\ I(w_{u2}, p) & = & \mathbf{false} \\ I(w_{u1}, q) & = & \mathbf{false} \\ I(w_{u2}, q) & = & \mathbf{true} \end{array}$$

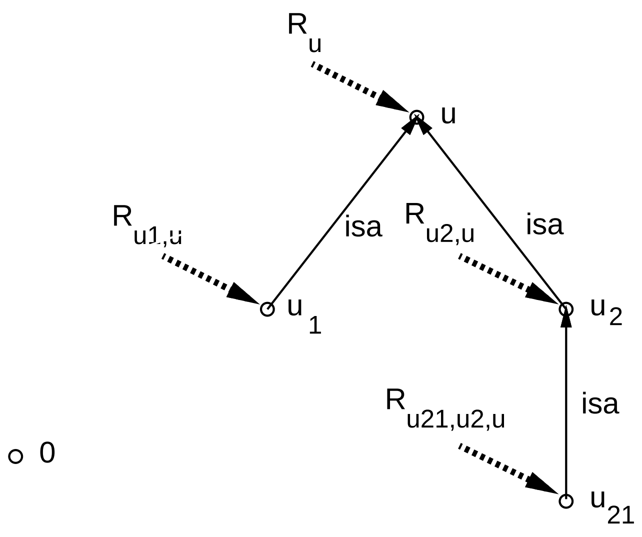

Figure 4.7: Example of a simple model—$\mathrm{MU}_{\mathcal{U}}$.

4.6.4 Resolution calculus

In order to present the resolution calculus for $\mathrm{MU}_{\mathcal{U}}$, we first must fix which formulae of the language of $\mathrm{MU}_{\mathcal{U}}$ we are going to treat as Horn clauses.

Definition 4.5 (Horn clauses) *Formulae $[v](p \leftarrow \Gamma, [\underline{z}]\Lambda)$, where p is an atomary formula, Γ and Λ are finite sets of atomary formulae, v is a unit, and $\underline{z}$ is a finite set of units, are called input clauses. Formulae $\leftarrow [\underline{u}]\Pi$, where Π is a finite set of atomary formulae and $\underline{u}$ is a finite set of units, are called goal statements.*

The resolution calculus for $\mathrm{MU}_{\mathcal{U}}$ contains one rule

- $\mathbf{R}_u^v$.

$$\frac{\leftarrow [u]p, [\underline{u^*}]\Pi \qquad [v](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

if u isa* v.

Theorem 4.11 (Soundness) *If a given goal statement $\leftarrow \mathcal{G}$ is resolvable to the empty goal statement by means of a given set $\mathcal{C}$ of input clauses, then it can be proved in* $\mathrm{MU}_{\mathcal{U}}$ *that $\mathcal{C} \to \mathcal{G}$.*

Proof. By induction on the height of the resolution tree.

1. Height=1.

$$\mathcal{C} \to \top.$$

2. Height > 1.

$$\dfrac{\dfrac{\mathcal{C} \to ([u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi) \ \text{ind. assumption}}{\mathcal{C} \to ([u]\Gamma, [u][\underline{z}]\Lambda, [\underline{u^*}]\Pi)}\ \mathbf{4}_{u,\underline{z}} \qquad \dfrac{\dfrac{\mathcal{C} \to [v](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\mathcal{C} \to [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\ \mathbf{K}^v_u}{\mathcal{C} \to ([u]p \leftarrow [u]\Gamma, [u][\underline{z}]\Lambda)}\ \mathbf{K}_u, \mathbf{C}_u}{\mathcal{C} \to ([u]p, [\underline{u^*}]\Pi)}$$

◁

Theorem 4.12 (Completeness) *If for a given set $\mathcal{C}$ of input clauses and for a given goal statement $\leftarrow \mathcal{G}$, it can be proved in* MU$_{\mathcal{U}}$ *that $\mathcal{C} \to \mathcal{G}$, then $\leftarrow \mathcal{G}$ is resolvable to the empty goal statement by means of $\mathcal{C}$.*

Proof. We show that from assuming that $\leftarrow \mathcal{G}$ is not resolvable to $\leftarrow$ by means of $\mathcal{C}$, it results that there exists a simple model which satisfies all the input clauses from $\mathcal{C}$ while not satisfying $\mathcal{G}$. Relying on Theorem 4.8, we then can conclude that $\mathcal{C} \to \mathcal{G}$ is not provable.

The simple model we need, is defined by the following condition:

$$I(u,p) = \textbf{true} \text{ iff } [u]p \text{ is resolvable to } \leftarrow \text{ by means of } \mathcal{C}.$$

Suppose that some formula $[u]p$ is not resolvable to $\leftarrow$. According to how we defined the model then $I(u,p) = \textbf{false}$. This evidently yields that $I(0,[u]p) = \textbf{false}$, which means that $[u]p$ is not satisfied by the model.

We assume that $\leftarrow \mathcal{G}$ is not resolvable to $\leftarrow$ by means of $\mathcal{C}$. Since $\mathcal{G}$ is a conjuction of formulae of the form $[u]p$, it is clear then that one of those formulae is not resolvable, and thus not satisfied by the model. Hence, the whole conjuction $\mathcal{G}$ is not satisfied by the model.

It now remains to prove that the input clauses $\mathcal{C}$ are satisfied by the model. Assume the contrary, i.e. that there is an input clause $[v](p \leftarrow \Gamma, [\underline{z}]\Lambda)$ in $\mathcal{C}$ which is not satisfied by the model. In this case, exists $w \in W$ such that

$$I(w, [v](p \leftarrow \Gamma, [\underline{z}]\Lambda)) = \textbf{false},$$

which means that there must exist $u \in U$ such that

$$I(u, (p \leftarrow \Gamma, [\underline{z}]\Lambda)) = \textbf{false}.$$

This is only possible if

$$\begin{aligned} I(u,p) &= \textbf{false}, \\ I(u,q) &= \textbf{true} \text{ for all } q \in \Gamma, \\ I(u,[z]r) &= \textbf{true} \text{ for all } [z]r \in [\underline{z}]\Lambda. \end{aligned}$$

The latter condition is equivalent to

$$I(z,r) = \textbf{true} \text{ for all } [z]r \in [\underline{z}]\Lambda.$$

Using the definition of our model, we on one hand conclude that $[u]p$ is not resolvable to $\leftarrow$. But on the other hand we similarly conclude that $[u]\Gamma$ and $[\underline{z}]\Lambda$ are resolvable to $\leftarrow$, which together with noticing that resolving $\leftarrow [u]p$ with $[v](p \leftarrow \Gamma, [\underline{z}]\Lambda)$ gives $\leftarrow [u]\Gamma, [\underline{z}]\Lambda$, means that $[u]p$ still is resolvable to $\leftarrow$. Contradiction.

◁

4.6.5 The predicate case

In the three previous sections, we studied the propositional fragment $\mathrm{MU}_{\mathcal{U}}$. It is obvious, however, that for practical applications this fragment is too poor. In almost all practical examples, properties must be allowed to have parameters, i.e. there have to be predicate symbols and terms in the language. It will be also clear from the examples in Section 4.11 that often it is desirable to allow arbitrary terms instead of only constants (unit names) as the parameters of modalities.

In the present thesis, we nevertheless study only the propositional fragments of the "OO logics" formally, since this is enough to clarify the principles, and saves us from too many overhead in the presentation. In examples, predicate languages are used. We limit the formal consideration of the full predicate logics to the following general comments.

We consider it natural to require traditionally that in the semantics individual symbols and function symbols be rigid, and the domain be constant. The majority of the theorems that we prove or hint at throughout the presentation then hold in the full logics. There are no big problems in the Kripke semantics or in the resolution calculi for the predicate logics (the resolution rule for a full logic is obtained from that of the corresponding propositional fragment by adding the unification step). As regards to the axiomatic calculi, the following can be said:

- The sound and complete Hilbert-style calculus for a given full logic can be obtained by adding the so-called *Barcan formula*

  $$\forall x \Box A \to \Box \forall x A$$

  as an axiom to the rules and axioms of the classical predicate Hilbert-style calculus and to the modal rules and axioms of the corresponding propositional fragment.

- There exist no brilliant Gentzen-style sequential calculi for the full logics. This is essentially due to Barcan formula which cannot be hidden into an introduction rule of some logical operator.

- The sound and complete Ohlbach-style calculus for a given full logic can be obtained simply by adding the introduction rules of quantifiers to the Ohlbach-style calculus of the corresponding propositional fragment.

4.6.6 Discussion

We mentioned at the end of Subsection 4.6.3 that not every formula which is satisfied by all the simple models, is provable. This might seem a defect of $\mathrm{MU}_{\mathcal{U}}$, since simple models closely correspond to the intuitive semantics of units. We now show that, in fact, this "defect" is natural, and cannot thus be recovered.

According to Theorems 4.8, 4.10, provability in $\mathrm{MU}_{\mathcal{U}}$ is equivalent to being satisfied by all the standard models of $\mathrm{MU}_{\mathcal{U}}$, and this is a much stricter condition than being satisfied by all the simple models of $\mathrm{MU}_{\mathcal{U}}$. The difference between standard and simple models is that in simple models there is exactly one world per each unit, whereas in standard models there can be several. It might seem good therefore to augment the restriction set on the accessibility relations in the Kripke semantics of $\mathrm{MU}_{\mathcal{U}}$ by the following restriction:

U$_u$-M. uniqueness : for any $w, w', w'' \in W$, if wR_uw' and wR_uw'' then $w' = w''$.

We prove that adding the restriction $\mathbf{U}_u$-M into the Kripke semantics of some logic yields an additional theorem

$$\mathbf{U}_u.\quad \langle u\rangle A \rightarrow [u]A$$

in its complete calculus.

Theorem 4.13 *The schema* $\mathbf{U}_u$ *is satisfied for all the models which meet the restriction* $\mathbf{U}_u$-M.

Proof. (We skip the index u.)

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall ww'w''(wRw'\&wRw'' \rightarrow w' = w'')} \\
\equiv & \forall ww'(wRw' \rightarrow \forall w''(wRw'' \rightarrow w' = w'')) \\
\rightarrow & \forall ww'(wRw'\&I(w',A) = \mathbf{true} \rightarrow \forall w''(wRw'' \rightarrow w' = w''\&I(w',A) = \mathbf{true})) \\
\rightarrow & \forall ww'(wRw'\&I(w',A) = \mathbf{true} \rightarrow \forall w''(wRw'' \rightarrow I(w'',A) = \mathbf{true})) \\
\equiv & \forall w(\exists w'(wRw'\&I(w',A) = \mathbf{true}) \rightarrow \forall w''(wRw'' \rightarrow I(w'',A) = \mathbf{true})) \\
\equiv & \forall w(I(w,\Diamond A) = \mathbf{true} \rightarrow I(w,\Box A) = \mathbf{true}) \\
\equiv & \forall w\; I(w,\Diamond A \rightarrow \Box A) = \mathbf{true}
\end{array}$$

◁

At the first glance it seems acceptable to add the formula $\mathbf{U}_u$ as an axiom to the Hilbert-style calculus of MU$_{\mathcal{U}}$. And its following equivalent formulation seems even more persuasive:

$$[u]\neg A \vee [u]A.$$

An axiom $\mathbf{U}_u$ would allow us to prove that a box distributes over a disjunction:

$$\begin{array}{c}
[u](\neg A \rightarrow B) \rightarrow ([u]\neg A \rightarrow [u]B)\ \mathbf{K}_u \\ \hline
[u](A \vee B) \rightarrow (\langle u\rangle A \vee [u]B) \\ \hline
[u](A \vee B) \rightarrow ([u]A \vee [u]B)\ \mathbf{U}^u
\end{array}$$

Unfortunately, $\mathbf{U}_u$ as axiom of MU$_{\mathcal{U}}$ has some undesirable side effects. Assume that u' isa u, and we get the following proof in MU$_{\mathcal{U}}$:

$$\begin{array}{c}
[u]\neg A \rightarrow [u']\neg A\ \mathbf{K}^u_{u'} \\ \hline
[u]\neg A \rightarrow \langle u'\rangle\neg A\ \mathbf{D}_{u'} \\ \hline
\langle u\rangle\neg A \rightarrow \langle u'\rangle\neg A\ \mathbf{U}^u \\ \hline
[u']A \rightarrow [u]A
\end{array}$$

Together with $\mathbf{K}^u_{u'}$, this implies $[u']A \equiv [u]A$ which makes it senseless to have units in isa relation, since the sets of formulae that hold in them, cannot differ at all.

The reason behind the unpleasant surprise is that we did not take into consideration the intended reading of $[u]A$ which was "A inheritably in u". $\langle u\rangle A$, accordingly, had to be read as "it does not hold that $\neg A$ inheritably in u". If $\langle u\rangle A$, it is possible that the cause for $\neg A$ not to hold inheritably in u, is that A holds in some unit u' isa* u, which does tell us nothing about whether A holds in u (inheritably or not), i.e. nothing about whether $[u]A$.

The argument can be reformulated as follows. Suppose it does not hold that A inheritably in u. If we had a right to conlude from this that $\neg A$ inheritably holds in u, we would reach at the conclusion that A can hold in no u' isa* u, which clearly would not be what we do want.

In terms of the simple models, what would happen when accepting $\mathbf{U}_u$, is that all the worlds accessible from somewhere by R_u, i.e. the worlds that correspond to some unit u' isa* u, would collapse into one.

A partial solution to the discussed "defect" will be given in the next section.

Another comment concerns the world 0 of simple models. It might seem that 0 should contain some universal knowledge that all the units could use (say, built-in predicates or like). So it might be proposed that the formula

$$\mathbf{U}^*_\mathbf{u}.\quad A \to [u]A$$

be added as an axiom to the Hilbert-style calculus of $\mathrm{MU}_\mathcal{U}$. But if we do accept $\mathbf{U}^*_\mathbf{u}$, we get the following proof in $\mathrm{MU}_\mathcal{U}$:

$$\dfrac{\dfrac{\neg A \to [u]\neg A \ \mathbf{U}^*_\mathbf{u} \qquad [u]\neg A \to \langle u\rangle\neg A \ \mathbf{D}_u}{\neg A \to \langle u\rangle \neg A}}{[u]A \to A}$$

Together with $\mathbf{U}^*_\mathbf{u}$, this implies that $[u]A \equiv A$ which is even worse than the result we got when trying to incorporate $\mathbf{U}_u$ into $\mathrm{MU}_\mathcal{U}$, since now for arbitrary $u, u' \in U$, we will have that $[u]A \equiv A \equiv [u']$. In terms of the simple models, it means that we tried to collapse all the worlds into one.

The correct way to look at 0 would be to take it, as if it were some observer who is external w.r.t. the unit hierarchy, and who can see everything that happens within it, while the unit hierachy itself even doesn't know that he/she is there. It might seem that the rule $\mathbf{N}_u$ still relates 0 and u, but this is not so. In terms of the simple models, this rule merely says that if 0 can prove A, then u can prove A. In other words, one is allowed to conclude that A holds in u, referring to 0, only if A is a logically inevitable truth and not something that holds just occasionally.

4.7 $\mathrm{MU}'_\mathcal{U}$—Inheritability Not Inevitable

The logic $\mathrm{MU}'_\mathcal{U}$ which we will describe in this section, is an advanced variant of $\mathrm{MU}_\mathcal{U}$, where the restriction that "everything be inheritable" has been removed.

Given a unit hierarchy $\mathcal{U} = \langle U,\ \text{isa}\ \rangle$, $\mathrm{MU}'_\mathcal{U}$ is a normal multimodal logic, where for each $u \in U$ there are two parameters—u and $u\downarrow$. The formula $[u]A$ is read as "A in u", whereas the formula $[u\downarrow]A$ is read as "A inheritably in u". Note that $[u]$ of $\mathrm{MU}_\mathcal{U}$ corresponds to $[u\downarrow]$ of $\mathrm{MU}'_\mathcal{U}$, not to $[u]$ of $\mathrm{MU}'_\mathcal{U}$, as it might be expected.

The organization of this section roughly repeats that of the previous one. Only the presentation is more compact—many proofs are omitted.

4.7.1 Motivation

In many cases, the assumption that everything is inheritable, turns out too strong. It might occur necessary to allow units own knowledge, which is not subject to inheritance. To this end, we develop a logic $\mathrm{MU}'_{\mathcal{U}}$ below, where for each unit we have two parameters—u and $u\downarrow$. $[u]A$ will mean "A in u", whereas $[u\downarrow]A$ will mean "A inheritably in u" (corresponding to what in $\mathrm{MU}_{\mathcal{U}}$ was denoted as $[u\downarrow]A$).

If $[u]A$ holds, it is guaranteed that A holds in u, and nothing is asserted about whether A holds in other units. Hence, it is evident that $[u\downarrow]A$ is stronger statement than $[u]A$, and this predicts the following axiom for $\mathrm{MU}'_{\mathcal{U}}$:

$$\mathbf{K}_u^{u\downarrow}.\quad [u\downarrow]A \to [u]A.$$

Since $[u\downarrow]$ corresponds to $[u]$ in $\mathrm{MU}_{\mathcal{U}}$, the axiom $\mathbf{K}_u^v$ of $\mathrm{MU}_{\mathcal{U}}$ maps to the following axiom in $\mathrm{MU}'_{\mathcal{U}}$:

$$\mathbf{K}_{u\downarrow}^{v\downarrow}.\quad [v\downarrow]A \to [u\downarrow]A \quad \text{if} \quad u \text{ isa } v.$$

In Subsection 4.6.6, we tried to supplement the axiomatics of $\mathrm{MU}_{\mathcal{U}}$ with the formula $\mathbf{U}_u$. The circumstance that in $\mathrm{MU}'_{\mathcal{U}}$ it is possible to speak about formulas that guaranteedly hold for one unit, removes the reasons that made this impossible, and in $\mathrm{MU}'_{\mathcal{U}}$, we will make $\mathbf{U}_u$ an axiom.

In Horn clauses, in $\mathrm{MU}_{\mathcal{U}}$ we allowed one modality over the whole clause, and optional modalities over the literals in the body. The modality over the whole clause pointed to the owner of the clause, whereas modalized literals in the body meant messages. In $\mathrm{MU}'_{\mathcal{U}}$, we will allow clauses to be inheritable or to hold guaranteedly only just in their owners, and accordingly, arbitrary parameters will be allowed in clause modalities. In message sending, on the contrary, we only require that the receiver be able to demonstrate the goal, and do not demand its descendants to be able to do the same. Parameters of goal modalities, therefore, will be limited to units.

As a final remark, we stress that $[u](p \to \Gamma)$ does not mean that a "code" Γ of a property p is private for the unit u. Although this "code" is inaccessible implicitly through inheritance, it still can be accessed explicitly via messages.

4.7.2 Axiomatics

The *Hilbert-style calculus* of $\mathrm{MU}'_{\mathcal{U}}$ consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Hilbert-style calculus.**
- **Modal rules and axioms** For each $u, v, z \in U$ we have: ($\circ$ and $\bullet$ can be either $\downarrow$ or the empty word)

$$\begin{array}{ll}
\mathbf{N}_{u\circ}. & \dfrac{A}{[u\circ]A}\,, \\
\mathbf{K}_{u\circ}. & [u\circ](A \to B) \to ([u\circ]A \to [u\circ]B), \\
\mathbf{D}_u. & [u]A \to \langle u\rangle A, \\
\mathbf{U}_u. & \langle u\rangle A \to [u]A, \\
\mathbf{K}_u^{u\downarrow}. & [u\downarrow]A \to [u]A, \\
\mathbf{K}_{u\downarrow}^{v\downarrow}. & [v\downarrow]A \to [u\downarrow]A \quad \text{if} \quad u \text{ isa } v, \\
\mathbf{4}_{u\bullet,z\circ}. & [z\circ]A \to [u\bullet][z\circ]A, \\
\mathbf{5}_{u\bullet,z\circ}. & \langle z\circ\rangle A \to [u\bullet]\langle z\circ\rangle A.
\end{array}$$

Theorem 4.14 *For each $u, v \in U$, if u isa* v then the formula*

$$\mathbf{K}^{v\downarrow}_{u\downarrow}.\quad [v\downarrow]A \to [u\downarrow]A$$

is provable in $\mathrm{MU}'_{\mathcal{U}}$.

Proof. Analogous to the proof of Theorem 4.1.

◁

Theorem 4.15 *For each $u \in U$, the formula*

$$\mathbf{D}_{u\downarrow}.\quad [u\downarrow]A \to \langle u\downarrow\rangle A$$

is provable in $\mathrm{MU}'_{\mathcal{U}}$.

Proof.

$$\dfrac{\dfrac{[u\downarrow]A \to [u]A\ ^{\mathbf{K}^{u\downarrow}_{u}} \qquad [u]A \to \langle u\rangle A\ ^{\mathbf{D}_u}}{[u\downarrow]A \to \langle u\rangle A} \qquad \dfrac{[u\downarrow]\neg A \to [u]\neg A\ ^{\mathbf{K}^{u\downarrow}_{u}}}{\langle u\rangle A \to \langle u\downarrow\rangle A}}{[u\downarrow]A \to \langle u\downarrow\rangle A}$$

◁

Theorem 4.16 *Given a sequence μ of modalities and a modality* m, *the formula*

$$\mu\mathrm{m}A \equiv \mathrm{m}A$$

is provable in $\mathrm{MU}'_{\mathcal{U}}$.

Proof. Analogous to the proof of Theorem 4.2.

◁

The *Gentzen-style sequential calculus* for $\mathrm{MU}'_{\mathcal{U}}$ consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Gentzen-style sequential calculus.**
- **Modal rules.** We formulate them in two rules. Γ, Δ, Λ, Π, Θ and Υ denote finite sets of formulae, m denotes a modality, $\underline{\mathrm{m}}$ and $\underline{\mathrm{m}'}$ denote finite sets of modalities, $\underline{v}$ and $\underline{v'}$ denote finite sets of units, $\underline{v\circ}$ and $\underline{v'\bullet}$ denote finite sets of parameters.

  $\mathbf{m}^{\underline{v\circ}|\underline{v'\bullet}}_{u}$.

  $$\dfrac{\Theta, \Gamma, \underline{\mathrm{m}}\Lambda \to \Upsilon, \Delta, \underline{\mathrm{m}'}\Pi}{\langle u\rangle\Theta, [\underline{v\circ}]\Gamma, \underline{\mathrm{m}}\Lambda \to [u]\Upsilon, \langle\underline{v'\bullet}\rangle\Delta, \underline{\mathrm{m}'}\Pi}$$

  if u isa* $\underline{v}, \underline{v'}$.

  $\mathbf{m}^{\underline{v\downarrow}|\underline{v'\downarrow}}_{u\downarrow}$.

  $$\dfrac{\Theta, \Gamma, \underline{\mathrm{m}}\Lambda \to \Upsilon, \Delta, \underline{\mathrm{m}'}\Pi}{\langle u\downarrow\rangle\Theta, [\underline{v}\downarrow]\Gamma, \underline{\mathrm{m}}\Lambda \to [u\downarrow]\Upsilon, \langle\underline{v'}\downarrow\rangle\Delta, \underline{\mathrm{m}'}\Pi}$$

  if u isa* $\underline{v}, \underline{v'}$, and $\|\Theta \cup \Upsilon\| \leq 1$.

Theorem 4.17 *The Hilbert-style and the Gentzen-style sequential calculi of* $\mathrm{MU}'_{\mathcal{U}}$ *are equivalent.*

Proof. Omitted. It follows the schema of Theorem 4.4. To the axiom $\mathbf{U}_u$, omission of the restriction $\|\Theta \cup \Upsilon\| \leq 1$ from the rule $\mathbf{m}_u^{v\circ|v'\bullet}$ corresponds.

◁

For the presented Gentzen-style calculus, cut elimination does not hold.

In order to present the Ohlbach-style calculus for $\mathrm{MU}'_{\mathcal{U}}$, we first define *world paths* for $\mathrm{MU}'_{\mathcal{U}}$.

Definition 4.6 (World paths) *Assume that for each $u \in U$, we have a symbol c_u whom we call the world constant of the unit u, and a countable set $F_u = \{f_{u1}, f_{u2}, \ldots\}$ of symbols whose elements we call world variables of the unit u. Then:*

1. *0 is a world path.*
2. *Every world constant is a world path.*
3. *Every world variable is a world path.*

The calculus contains the following rules and axioms:

- **Labelled versions of the rules and axioms of the classical propositional Gentzen-style sequential calculus.**
- **Modal rules.** They are eight in number. g denotes a world path, $\underline{h}$ and $\underline{h'}$ denote finite sets of world paths, f denotes a non-zero world constant or variable, and $\mathrm{unit}(f)$ is the unit of f.

  $[v]\!\rightarrow$.

  $$\frac{(c_v)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)[v]A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

  $[v\downarrow]\!\rightarrow$.

  $$\frac{(f)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)[v\downarrow]A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

  where $\mathrm{unit}(f)$ isa* v.

  $\rightarrow\![u]$.

  $$\frac{(\underline{h})\Lambda \rightarrow (c_u)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)[u]A, (\underline{h'})\Pi}$$

  $\rightarrow\![u]$.

  $$\frac{(\underline{h})\Lambda \rightarrow (f)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)[u]A, (\underline{h'})\Pi}$$

where unit$(f) = u$ and f does not occur anywhere in the conclusion sequent and lower in the proof.

$\langle u \rangle \rightarrow$.

$$\frac{(c_u)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)\langle u \rangle A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

$\langle u \downarrow \rangle \rightarrow$.

$$\frac{(f)A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}{(g)\langle u \downarrow \rangle A, (\underline{h})\Lambda \rightarrow (\underline{h'})\Pi}$$

where unit$(f) = u$ and f does not occur anywhere in the conclusion sequent and lower in the proof.

$\rightarrow \langle v \rangle$.

$$\frac{(\underline{h})\Lambda \rightarrow (c_v)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)\langle v \rangle A, (\underline{h'})\Pi}$$

$\rightarrow \langle v \downarrow \rangle$.

$$\frac{(\underline{h})\Lambda \rightarrow (f)A, (\underline{h'})\Pi}{(\underline{h})\Lambda \rightarrow (g)\langle v \downarrow \rangle A, (\underline{h'})\Pi}$$

where unit(f) isa* v.

Theorem 4.18 *The Gentzen-style sequential and the Ohlbach-style calculi of* $\mathrm{MU}'_{\mathcal{U}}$ *are equivalent in the sense that a sequent* $\Lambda \rightarrow \Pi$ *is provable in Gentzen style iff the sequent* $(0)\Lambda \rightarrow (0)\Pi$ *is provable in Ohlbach style.*

Proof. Omitted. It follows the proof schema of Theorem 4.5.

$\lhd$

Theorem 4.19 *In the Ohlbach-style calculus for* $\mathrm{MU}'_{\mathcal{U}}$*, cuts can be eliminated.*

Proof. Omitted. It follows the proof schema of Theorem 4.6.

$\lhd$

Theorem 4.20 $\mathrm{MU}'_{\mathcal{U}}$ *is decidable.*

Proof. Omitted. It follows the proof schema of Theorem 4.7.

$\lhd$

4.7.3 Semantics

The restrictions on the accessibility relations in the model class (the Kripke semantics) of $\mathrm{MU}'_{\mathcal{U}}$ are the following:

$\mathbf{D}_u$-M.	seriality	: for any $w \in W$ there exist $w' \in W$ such that wR_uw',
$\mathbf{U}_u$-M.	uniqueness	: for any $w, w', w'' \in W$, if wR_uw' and wR_uw'' then $w' = w''$,
$\mathbf{K}_u^{u\downarrow}$-M.	inclusion	: for any $w, w' \in W$ if wR_uw' then $wR_{u\downarrow}w'$,
$\mathbf{K}_{u\downarrow}^{v\downarrow}$-M.	inclusion	: if u isa v then for any $w, w' \in W$ if $wR_{u\downarrow}w'$ then $wR_{v\downarrow}w'$,
$\mathbf{4}_{u^\bullet, z^\circ}$-M.	quasi-transitivity	: for any $w, w', w'' \in W$ if $wR_{u^\bullet}w'$ and $w'R_{z^\circ}w''$ then $wR_{z^\circ}w''$,
$\mathbf{5}_{u^\bullet, z^\circ}$-M.	quasi-euclidity	: for any $w, w', w'' \in W$ if $wR_{u^\bullet}w'$ and $wR_{z^\circ}w''$ then $w'R_{z^\circ}w''$.

Theorem 4.21 (Soundness) *If a formula is provable in* $\mathrm{MU}'_{\mathcal{U}}$ *then it is valid in the Kripke semantics of* $\mathrm{MU}'_{\mathcal{U}}$*, i.e. it is satisfied by every model meeting the restrictions* $\mathbf{D}_u$-M, $\mathbf{U}_u$-M, $\mathbf{K}_u^{u\downarrow}$-M, $\mathbf{K}_{u\downarrow}^{v\downarrow}$-M, $\mathbf{4}_{u,z}$-M, *and* $\mathbf{5}_{u,z}$-M. *Moreover, none of these restrictions can be left out.*

Proof. Omitted. It follows the proof schema of Theorem 4.8. It also uses Theorem 4.13.

$\lhd$

Definition 4.7 (Standard models) *Assume we are given a unit hierarchy* $\mathcal{U} = \langle U, \text{ isa } \rangle$. *Then we say that a Kripke model* $\langle W, \{R_u, R_{u\downarrow} : u \in U\}, I\rangle$ *is a standard model of* $\mathrm{MU}'_{\mathcal{U}}$*, if:*

- $W = \{0\} \cup \bigcup_{u \in U} W_u$*, where* W_u*'s are pairwise non-intersecting non-empty finite sets, each having one distinguished element* w_u*;*
- *and for any* $w, w' \in W$ *and* $u \in U$*,*

$$\begin{array}{rll} wR_uw' & \textit{iff} & w' = w_u, \\ wR_{u\downarrow}w' & \textit{iff} & \textit{there exists } u' \in U \textit{ such that } w' \in W'_u \textit{ and } u' \text{ isa}^* u. \end{array}$$

Theorem 4.22 *For any* $\mathcal{U}$*, the standard models of* $\mathrm{MU}'_{\mathcal{U}}$ *belong to the model class of* $\mathrm{MU}'_{\mathcal{U}}$.

Proof. Omitted. It follows the proof schema of Theorem 4.9.

$\lhd$

Theorem 4.23 (Completeness) *If a formula is valid in the Kripke semantics of* $\mathrm{MU}'_{\mathcal{U}}$*, i.e. if it is satisfied by every model meeting the restrictions* $\mathbf{D}_u$-M, $\mathbf{U}_u$-M, $\mathbf{K}_u^{u\downarrow}$-M, $\mathbf{K}_{u\downarrow}^{v\downarrow}$-M, $\mathbf{4}_{u,z}$-M, *and* $\mathbf{5}_{u,z}$-M, *then it is provable in* $\mathrm{MU}'_{\mathcal{U}}$.

Proof. Omitted. It follows the proof schema of Theorem 4.10.

$\lhd$

For $\mathrm{MU}'_{\mathcal{U}}$, the definition of *simple models* takes the following form:

Definition 4.8 (Simple models) *Assume we are given a unit hierarchy $\mathcal{U} = \langle \mathcal{U}, \text{ isa } \rangle$. Then we say that a Kripke model $\langle W, \{R_u, R_{u\downarrow} : u \in U\}, I\rangle$ is a simple model of* $\mathrm{MU}'_{\mathcal{U}}$, *if:*

- *$W = \{0\} \cup U$,*
- *and for any $w, w' \in W$ and $u \in U$*

$$\begin{aligned} wR_u w' \quad &\textit{iff} \quad w' = u, \\ wR_{u\downarrow} w' \quad &\textit{iff} \quad w' \in U \textit{ and } w' \text{ isa}^* \ u. \end{aligned}$$

Again, obviously, every simple model is standard model. In simple models, in full accordance with intuition, we have that:

$I(w, [u]A) = \mathbf{true}$ iff $I(u, A) = \mathbf{true}$,
$I(w, [u \downarrow]A) = \mathbf{true}$ iff for all $w' \in U$ such that w' isa* u, $I(w', A) = \mathbf{true}$.

The relationship between isa and accessibility relations in simple models is illustrated by the example in Figure 4.8.

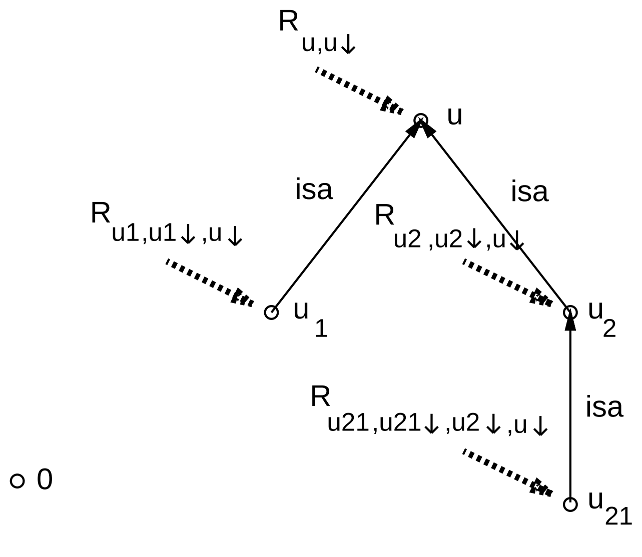
$R_{u,u\downarrow}$
u
isa
isa
$R_{u1,u1\downarrow,u\downarrow}$
$R_{u2,u2\downarrow,u\downarrow}$
u_1
u_2
$R_{u21,u21\downarrow,u2\downarrow,u\downarrow}$
isa
0
u_{21}

Figure 4.8: Example of a simple model—$\mathrm{MU}'_{\mathcal{U}}$.

4.7.4 Resolution calculus

For $\mathrm{MU}'_{\mathcal{U}}$, we define Horn clauses in the following way:

Definition 4.9 (Horn clauses) *Formulae $[v\circ](p \leftarrow \Gamma, [\underline{z}]\Lambda)$, where p is an atomary formula, Γ and Λ are finite sets of atomary formulae, $v\circ$ is a parameter, and $\underline{z}$ is a finite set of units, are called input clauses. Formulae $\leftarrow [\underline{u}]\Pi$, where Π is a finite set of atomary formulae, and $\underline{u}$ is a finite set of units, are called goal statements.*

The resolution calculus for $\mathrm{MU}'_{\mathcal{U}}$ contains two rules:

- $\mathbf{R}_u$.

$$\frac{\leftarrow [u]p, [\underline{u^*}]\Pi \qquad [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

- $\mathbf{R}_u^{v\downarrow}$.

$$\frac{\leftarrow [u]p, [\underline{u^*}]\Pi \qquad [v\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

  where u isa* v.

Theorem 4.24 (Soundness) *If a given goal statement $\leftarrow \mathcal{G}$ is resolvable to the empty goal statement by means of a given set $\mathcal{C}$ of input clauses, then it can be proved in $\mathrm{MU}'_{\mathcal{U}}$ that $\mathcal{C} \rightarrow \mathcal{G}$.*

Proof. By induction on the height of the resolution tree.

1. Height=1.

$$\mathcal{C} \rightarrow \top.$$

2. Height > 1.

- $\mathbf{R}_u$.

$$\dfrac{\dfrac{\mathcal{C} \rightarrow ([u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \rightarrow ([u]\Gamma, [u][\underline{z}]\Lambda, [\underline{u^*}]\Pi)}\,\mathbf{4}_{u,\underline{z}} \qquad \dfrac{\mathcal{C} \rightarrow [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)\ \mathbf{K}_u^v}{\mathcal{C} \rightarrow ([u]p \leftarrow [u]\Gamma, [u][\underline{z}]\Lambda)}\,\mathbf{K}_u,\mathbf{C}_u}{\mathcal{C} \rightarrow ([u]p, [\underline{u^*}]\Pi)}$$

- $\mathbf{R}_u^{v\downarrow}$.

$$\dfrac{\dfrac{\mathcal{C} \rightarrow ([u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \rightarrow ([u]\Gamma, [u][\underline{z}]\Lambda, [\underline{u^*}]\Pi)}\,\mathbf{4}_{u,\underline{z}} \qquad \dfrac{\dfrac{\dfrac{\mathcal{C} \rightarrow [v\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\mathcal{C} \rightarrow [u\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\,\mathbf{K}_{u\downarrow}^{v\downarrow}}{\mathcal{C} \rightarrow [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\,\mathbf{K}_u^{u\downarrow}}{\mathcal{C} \rightarrow ([u]p \leftarrow [u]\Gamma, [u][\underline{z}]\Lambda)}\,\mathbf{K}_u,\mathbf{C}_u}{\mathcal{C} \rightarrow ([u]p, [\underline{u^*}]\Pi)}$$

◁

Theorem 4.25 (Completeness) *If for a given set $\mathcal{C}$ of input clauses and for a given goal statement $\leftarrow \mathcal{G}$, it can be proved in $\mathrm{MU}'_{\mathcal{U}}$ that $\mathcal{C} \rightarrow \mathcal{G}$, then $\leftarrow \mathcal{G}$ is resolvable to the empty goal statement by means of $\mathcal{C}$.*

Proof. The proof of Theorem 4.12 goes through with almost no changes.

◁

4.7.5 Discussion

As we had to experience in Subsection 4.7.3, though the axiom $\mathbf{U}_u$ caused the standard models of $\mathrm{MU}'_{\mathcal{U}}$ to have a distinguished world for each unit, it still did not collapse the concept of standard model to that of (intuitive) simple model, since non-distinguished worlds were not excluded.

Due to this, the example for $\mathrm{MU}_{\mathcal{U}}$ from the end of Subsection 4.6.3 of a formula that is satisfied by all simple models but unprovable, applies for $\mathrm{MU}'_{\mathcal{U}}$, too, except that in $\mathrm{MU}'_{\mathcal{U}}$, it must be written as:

$$[u\downarrow](p \vee q) \rightarrow [u\downarrow]p \vee [u\downarrow]q.$$

The way to get round of this would be to introduce the following restriction on the accessibility relations, which would forbid the non-distinguished worlds:

$\mathbf{F}_{u\downarrow}$-M. finishedness : for any $w, w' \in W$, if $wR_{u\downarrow}w'$ then there exists $u' \in U$ such that u' isa* u and $wR_{u'}w'$.

We prove that adding the restriction $\mathbf{F}_{u\downarrow}$-M into the Kripke semantics of some logic yields an additional theorem

$$\mathbf{F}_{u\downarrow}. \quad \bigwedge_{u' \text{ isa}^* u} [u']A \rightarrow [u\downarrow]A$$

in its complete calculus.

Theorem 4.26 *The schema $\mathbf{F}_{u\downarrow}$ is satisfied for all the models which meet the restriction $\mathbf{F}_{u\downarrow}$-M.*

Proof.

$$\begin{aligned}
&\forall ww'[wR_{u\downarrow}w' \rightarrow (\exists u' \text{ isa}^* u)wR_{u'}w']\\
\equiv\;& \forall ww'[wR_{u\downarrow}w' \& \neg I(w',A) = \mathbf{true} \rightarrow (\exists u' \text{ isa}^* u)wR_{u'}w' \& \neg I(w',A) = \mathbf{true}]\\
\equiv\;& \forall ww'[\neg(wR_{u\downarrow}w' \rightarrow I(w',A) = \mathbf{true}) \rightarrow \neg(\forall u' \text{ isa}^* u)(wR_{u'}w' \rightarrow I(w',A) = \mathbf{true})]\\
\equiv\;& \forall ww'[(\forall u' \text{ isa}^* u)(wR_{u'}w' \rightarrow I(w',A) = \mathbf{true}) \rightarrow (wR_{u\downarrow}w' \rightarrow I(w',A) = \mathbf{true})]\\
\rightarrow\;& \forall w[(\forall u' \text{ isa}^* u)\forall w'(wR_{u'}w' \rightarrow I(w',A) = \mathbf{true}) \rightarrow \forall w'(wR_{u\downarrow}w' \rightarrow I(w',A) = \mathbf{true})]\\
\equiv\;& \forall w[(\forall u' \text{ isa}^* u)I(w,[u']A) = \mathbf{true} \rightarrow I(w,[u\downarrow]A) = \mathbf{true}]\\
\equiv\;& \forall w\; I(w, (\bigwedge_{u' \text{ isa}^* u} [u']A \rightarrow [u\downarrow]A)) = \mathbf{true}
\end{aligned}$$

$\triangleleft$

Though $\mathbf{F}_{u\downarrow}$ as an additional axiom would collapse the concept of standard model to the concept of simple model, we still do not want to make it axiom. The reason is that the theory $\mathrm{MU}'_{\mathcal{U}}$ so far has been monotonic w.r.t. extensions of $\mathcal{U}$, i.e. that for any extension $\mathcal{U}_e$ of $\mathcal{U}$, the theory $\mathrm{MU}'_{\mathcal{U}}$ has been a subset of the theory $\mathrm{MU}'_{\mathcal{U}_e}$.

$\mathbf{F}_{u\downarrow}$ as axiom would spoil this monotonicity. Indeed, assume we are given a unit hierarchy $\mathcal{U} = \langle U, \text{isa}\rangle$ and a unit $u \in U$. Consider the unit hierarchy $\mathcal{U}_e = \langle U_e, \text{isa}_e\rangle$ where $U_e = U \cup \{u_e\}$ and $\text{isa}_e = \text{isa} \cup \{\langle u_e, u\rangle\}$. In $\mathrm{MU}'_{\mathcal{U}_e}$, an exemplar

$$\bigwedge_{u' \text{ isa}^* u} [u']p \rightarrow [u\downarrow]p$$

of $\mathbf{F}_{u\downarrow}$ of MU$'_{\mathcal{U}}$ is not provable, since it is not satisfied by the following simple model of MU$'_{\mathcal{U}_e}$:

$$\begin{aligned} I(u',p) &= \textbf{true} \text{ if } u' \text{ isa}^* \ u, \\ I(u_e,p) &= \textbf{false}. \end{aligned}$$

Concluding the discussion of the logic MU$'_{\mathcal{U}}$, we wish to mention that there seems to exist another approach to building up MU$'_{\mathcal{U}}$, where the only parameters would be units and the symbol $\downarrow$. This approach has not been studied carefully yet, but it seems that the following set of modal rules and axioms would do:

$$\begin{array}{ll}
\mathbf{N}_u. & \dfrac{A}{[u]A}\,, \\[2ex]
\mathbf{N}_\downarrow. & \dfrac{A}{[\downarrow]A}\,, \\[2ex]
\mathbf{K}_u. & [u](A \to B) \to ([u]A \to [u]B), \\
\mathbf{K}_\downarrow. & [\downarrow](A \to B) \to ([\downarrow]A \to [\downarrow]B), \\
\mathbf{D}_u. & [u]A \to \langle u\rangle A, \\
\mathbf{U}_u. & \langle u\rangle A \to [u]A, \\
\mathbf{T}_\downarrow. & [\downarrow]A \to A, \\
\mathbf{K}^{v\downarrow}_{u\downarrow}. & [v][\downarrow]A \to [u][\downarrow]A \quad \text{if} \quad u \text{ isa } v, \\
\mathbf{4}_{u,z}. & [z]A \to [u][z]A, \\
\mathbf{4}_{\downarrow,z}. & [z]A \to [\downarrow][z]A, \\
\mathbf{4}_{\downarrow,\downarrow}. & [\downarrow]A \to [\downarrow][\downarrow]A, \\
\mathbf{5}_{u,z}. & \langle z\rangle A \to [u]\langle z\rangle A, \\
\mathbf{5}_{\downarrow,z}. & \langle z\rangle A \to [\downarrow]\langle z\rangle A.
\end{array}$$

To the former modalities of the form $[u\downarrow]$, the composition $[u][\downarrow]$ would correspond in this approach. Every $\downarrow$ essentially belongs to the closest unit leftwards it. Iterated $\downarrow$'s are equivalent to one $\downarrow$. There is no axiom $\mathbf{5}_{\downarrow,\downarrow}$, to avoid $R_\downarrow$ become symmetric in the semantics.

The relationship between isa and accessibility relations in simple models in the described approach is illustrated by the example in Figure 4.9.

4.8 MU$''_{\mathcal{U}}$—Inheriting Not Inevitable

The logic MU$'_{\mathcal{U}}$ made it possible to block inheritance "from above", i.e. to say of a formula that holds in a unit, whether it holds inheritably, or its holding is guaranteed just for this particular unit. The logic MU$''_{\mathcal{U}}$ whom we describe in this section, enables also dual blocking "from below", allowing to say of a formula that holds in a unit, whether its holding can be derived without relying on inheriting, or it just holds.

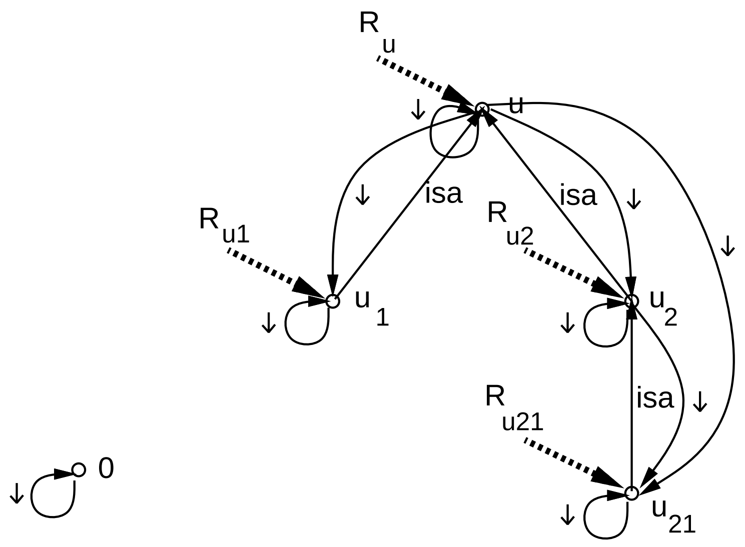

Figure 4.9: Example of a simple model—$\mathrm{MU}'_{\mathcal{U}}$, alternative approach.

Given a unit hierarchy $\mathcal{U} = \langle U, \text{ isa } \rangle$, $\mathrm{MU}'_{\mathcal{U}}$ is a normal multimodal logic, where for each $u \in U$ there are four parameters—u, $u \downarrow$, $u \uparrow$, $u \downarrow\uparrow$. Modal formulae are read as follows:

$[u]A$	reads as	"A, without using inheritance, in u",
$[u \downarrow]A$	reads as	"A, without using inheritance, inheritably in u",
$[u \uparrow]A$	reads as	"A in u",
$[u \downarrow\uparrow]A$	reads as	"A inheritably in u.

Note that $[u \uparrow]$ of $\mathrm{MU}''_{\mathcal{U}}$ corresponds to $[u]$ of $\mathrm{MU}'_{\mathcal{U}}$, and that $[u \downarrow\uparrow]$ of $\mathrm{MU}''_{\mathcal{U}}$ corresponds to $[u \downarrow]$ of $\mathrm{MU}'_{\mathcal{U}}$ and to $[u]$ of $\mathrm{MU}_{\mathcal{U}}$.

At the present stage, the logic $\mathrm{MU}''_{\mathcal{U}}$ has not been studied as carefully as $\mathrm{MU}_{\mathcal{U}}$ and $\mathrm{MU}'_{\mathcal{U}}$. By organization, this section is similar that of the two previous ones, though it considers no axiomatics other than the Hilbert-style one, and does not consider Kripke semantics.

4.8.1 Motivation

Like $\mathrm{MU}'_{\mathcal{U}}$ allowed to block inheritance "from above", it is also thinkable to consider blocking "from below". To try this, we develop a logic $\mathrm{MU}''_{\mathcal{U}}$ below, where for each unit, there are four parameters—u, $u \downarrow$, $u \uparrow$, $u \downarrow\uparrow$, and modal formulae are read as follows:

$[u]A$	reads as	"A, without using inheritance, in u",
$[u \downarrow]A$	reads as	"A, without using inheritance, inheritably in u",
$[u \uparrow]A$	reads as	"A in u",
$[u \downarrow\uparrow]A$	reads as	"A inheritably in u.

Note that the modalities of $\mathrm{MU}'_{\mathcal{U}}$ get new, different notations in $\mathrm{MU}''_{\mathcal{U}}$.

From the readings it is clear that $[u]A$ and $[u\downarrow]A$ are stronger statements than $[u\uparrow]A$ and $[u\downarrow\uparrow]A$, respectively. This predicts us the following two axioms for $\mathrm{MU}''_{\mathcal{U}}$:

$$\begin{array}{ll} \mathbf{K}^{u}_{u\uparrow}. & [u]A \to [u\uparrow]A, \\ \mathbf{K}^{u\downarrow}_{u\downarrow\uparrow}. & [u\downarrow]A \to [u\downarrow\uparrow]A. \end{array}$$

Similarly, since "holding inheritably" involves more than "just holding" (as we observed in Subsection 4.7.1), $[u\downarrow]A$ and $[u\downarrow\uparrow]A$ are stronger statements than $[u]A$ and $[u\uparrow]A$, which predicts us another pair of axioms for $\mathrm{MU}''_{\mathcal{U}}$:

$$\begin{array}{ll} \mathbf{K}^{u\downarrow}_{u}. & [u\downarrow]A \to [u]A, \\ \mathbf{K}^{u\downarrow\uparrow}_{u\uparrow}. & [u\downarrow\uparrow]A \to [u\uparrow]A. \end{array}$$

The last axiom is the correspondent to the axiom $\mathbf{K}^{u\downarrow}_{u}$ of $\mathrm{MU}'_{\mathcal{U}}$.

The axioms $\mathbf{K}^{v\downarrow}_{u\downarrow}$ and $\mathbf{U}_u$ of $\mathrm{MU}'_{\mathcal{U}}$ take the following shapes in $\mathrm{MU}''_{\mathcal{U}}$:

$$\begin{array}{ll} \mathbf{K}^{v\downarrow\uparrow}_{u\downarrow\uparrow}. & [v\downarrow\uparrow]A \to [u\downarrow\uparrow]A \quad \text{if} \quad u \text{ isa } v, \\ \mathbf{U}_{u\uparrow}. & \langle u\uparrow\rangle A \to [u\uparrow]A. \end{array}$$

We do not postulate

$$\mathbf{U}_u.\ \ \langle u\rangle A \to [u]A$$

in $\mathrm{MU}''_{\mathcal{U}}$. The argument behind this is the following. Suppose that $\langle u\rangle A$, i.e. that it can't be derived without reference to inheriting, that $\neg A$ holds in u. This does not exclude that $\neg A$ might still hold in u, which would mean that A does not hold in u, and that the more it would not be derivable without reference to inheriting, that A holds in u. But $\mathbf{U}_u$ excludes this possibility.

In Horn clauses, in $\mathrm{MU}'_{\mathcal{U}}$, where inheritance could be blocked "from above", we allowed clause modalities either to claim inheritability or not, whereas goal modalities were not allowed to require inheritability. In $\mathrm{MU}''_{\mathcal{U}}$, with regards to blocking "from below", we will act in a dual manner. Namely, clause modalities will always have to claim holding without reference to inheriting (since input clauses are a kind of "initial knowledge"), whilst goal modalities will be allowed to require demonstration "just so" or without reference to inheriting.

The relationship between different modalities of $\mathrm{MU}''_{\mathcal{U}}$, and their usage in Horn clauses, is illustrated in Figure 4.10.

We have to admit that $\mathrm{MU}''_{\mathcal{U}}$ is more interesting for theory than for practical applications. We have not found counterparts to blocking "from below" in OO languages. The examples in Section 4.11 will not go further than $\mathrm{MU}'_{\mathcal{U}}$ either.

4.8.2 Axiomatics

The *Hilbert-style calculus* of $\mathrm{MU}''_{\mathcal{U}}$ consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Hilbert-style calculus.**

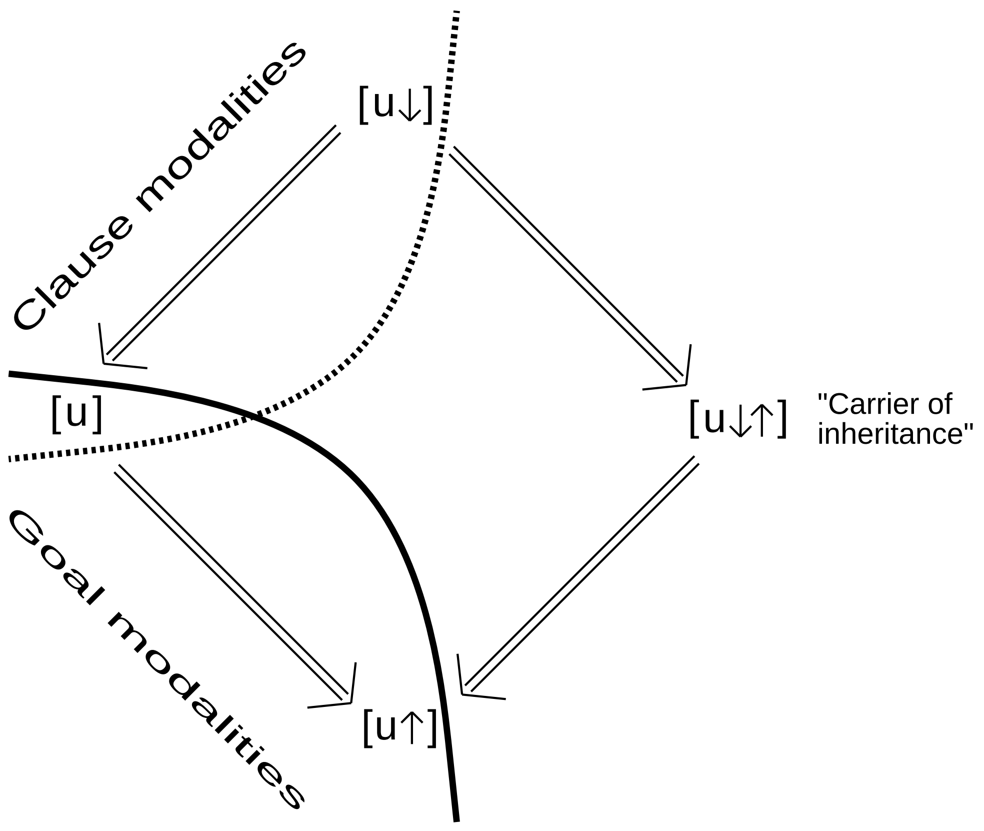

Figure 4.10: Modalities in $\mathrm{MU}''_{\mathcal{U}}$.

- **Modal rules and axioms** For each $u, v, z \in U$ we have: ($\circ$ and $\bullet$ can be either $\downarrow$, the empty word, $\downarrow\uparrow$ or $\uparrow$)

$$
\begin{array}{ll}
\mathbf{N}_{u\circ}. & \dfrac{A}{[u\circ]A}\ , \\
\mathbf{K}_{u\circ}. & [u\circ](A \to B) \to ([u\circ]A \to [u\circ]B), \\
\mathbf{D}_{u\uparrow}. & [u\uparrow]A \to \langle u\uparrow\rangle A, \\
\mathbf{U}_{u\uparrow}. & \langle u\uparrow\rangle A \to [u\uparrow]A, \\
\mathbf{K}^{u\downarrow}_{u}. & [u\downarrow]A \to [u]A, \\
\mathbf{K}^{u\downarrow\uparrow}_{u\uparrow}. & [u\downarrow\uparrow]A \to [u\uparrow]A, \\
\mathbf{K}^{u}_{u\uparrow}. & [u]A \to [u\uparrow]A, \\
\mathbf{K}^{u\downarrow}_{u\downarrow\uparrow}. & [u\downarrow]A \to [u\downarrow\uparrow]A, \\
\mathbf{K}^{v\downarrow\uparrow}_{u\downarrow\uparrow}. & [v\downarrow\uparrow]A \to [u\downarrow\uparrow]A \quad \text{if} \quad u \text{ isa } v, \\
\mathbf{4}_{u\bullet,z\circ}. & [z\circ]A \to [u\bullet][z\circ]A, \\
\mathbf{5}_{u\bullet,z\circ}. & \langle z\circ\rangle A \to [u\bullet]\langle z\circ\rangle A.
\end{array}
$$

Theorem 4.27 *For each $u, v \in U$, if u isa* v then the formula*

$$\mathbf{K}^{v\downarrow\uparrow}_{u\downarrow\uparrow}.\quad [v\downarrow\uparrow]A \to [u\downarrow\uparrow]A$$

is provable in $\mathrm{MU}''_{\mathcal{U}}$.

Proof. Analogous to the proof of Theorem 4.1.

◁

Theorem 4.28 *For each $u \in U$, the formulae*

$$\begin{array}{ll} \mathbf{D}_u. & [u]A \to \langle u\rangle A, \\ \mathbf{D}_{u\downarrow\uparrow}. & [u \downarrow\uparrow]A \to \langle u \downarrow\uparrow\rangle A, \\ \mathbf{D}_{u\downarrow}. & [u \downarrow]A \to \langle u \downarrow\rangle A \end{array}$$

are provable in $\mathrm{MU}''_{\mathcal{U}}$.

Proof. Analogous to the Proof of Theorem 4.15.

◁

Theorem 4.29 *Given a sequence μ of modalities and a modality* m, *the formula*

$$\mu \mathrm{m} A \equiv \mathrm{m} A$$

is provable in $\mathrm{MU}''_{\mathcal{U}}$.

Proof. Analogous to the proof of Theorem 4.2.

◁

4.8.3 Resolution calculus

For $\mathrm{MU}''_{\mathcal{U}}$, Horn clauses are defined as follows:

Definition 4.10 (Horn clauses) *Formulae $[v\circ](p \leftarrow \Gamma, [\underline{z}]\Lambda)$, where p is an atomary formula, Γ and Λ are finite sets of atomary formulae, v is a unit, $\circ$ is either empty word or $\downarrow$, and $\underline{z}$ is a finite set of parameters formed of only unit symbols and $\uparrow$'s, are called input clauses. Formulae $\leftarrow [\underline{u}]\Pi$, where Π is a finite set of atomary formulae, and $\underline{u}$ is a finite set of parameters formed of only unit symbols and $\uparrow$'s, are called goal statements.*

The resolution calculus for $\mathrm{MU}''_{\mathcal{U}}$ contains four rules:

- $\mathbf{R}_u$.

$$\frac{\leftarrow [u]p, [\underline{u^*}]\Pi \qquad [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

- $\mathbf{R}_u^{u\downarrow}$.

$$\frac{\leftarrow [u]p, [\underline{u^*}]\Pi \qquad [u \downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

- $\mathbf{R}^{u}_{u\uparrow}$.

$$\frac{\leftarrow [u\uparrow]p, [\underline{u^*}]\Pi \qquad [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u\uparrow]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

- $\mathbf{R}^{v\downarrow}_{u\uparrow}$.

$$\frac{\leftarrow [u\uparrow]p, [\underline{u^*}]\Pi \qquad [v\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\leftarrow [u\uparrow]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi}$$

where u isa* v.

Theorem 4.30 (Soundness) *If a given goal statement $\leftarrow \mathcal{G}$ is resolvable to the empty goal statement by means of a given set $\mathcal{C}$ of input clauses, then it can be proved in* $\mathrm{MU}''_{\mathcal{U}}$ *that $\mathcal{C} \rightarrow \mathcal{G}$.*

Proof. By induction on the height of the resolution tree.

1. Height=1.

$$\mathcal{C} \rightarrow \top.$$

2. Height > 1.

- $\mathbf{R}_{u}$.

$$\frac{\dfrac{\mathcal{C} \rightarrow ([u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \rightarrow ([u]\Gamma, [u][\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \mathbf{4}_{u,\underline{z}}} \qquad \dfrac{\mathcal{C} \rightarrow [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)\ \mathbf{K}^{v}_{u}}{\mathcal{C} \rightarrow ([u]p \leftarrow [u]\Gamma, [u][\underline{z}]\Lambda)\ \mathbf{K}_{u}, \mathbf{C}_{u}}}{\mathcal{C} \rightarrow ([u]p, [\underline{u^*}]\Pi)}$$

- $\mathbf{R}^{u\downarrow}_{u}$.

$$\frac{\dfrac{\mathcal{C} \rightarrow ([u]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \rightarrow ([u]\Gamma, [u][\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \mathbf{4}_{u,\underline{z}}} \qquad \dfrac{\dfrac{\mathcal{C} \rightarrow [u\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\mathcal{C} \rightarrow [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)\ \mathbf{K}^{u\downarrow}_{u}}}{\mathcal{C} \rightarrow ([u]p \leftarrow [u]\Gamma, [u][\underline{z}]\Lambda)\ \mathbf{K}_{u}, \mathbf{C}_{u}}}{\mathcal{C} \rightarrow ([u]p, [\underline{u^*}]\Pi)}$$

- $\mathbf{R}^{u}_{u\uparrow}$.

$$\frac{\dfrac{\mathcal{C} \rightarrow ([u\uparrow]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \rightarrow ([u\uparrow]\Gamma, [u\uparrow][\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \mathbf{4}_{u,\underline{z}}} \qquad \dfrac{\dfrac{\mathcal{C} \rightarrow [u](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\mathcal{C} \rightarrow [u\uparrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)\ \mathbf{K}^{u}_{u\uparrow}}}{\mathcal{C} \rightarrow ([u\uparrow]p \leftarrow [u\uparrow]\Gamma, [u\uparrow][\underline{z}]\Lambda)\ \mathbf{K}_{u}, \mathbf{C}_{u}}}{\mathcal{C} \rightarrow ([u\uparrow]p, [\underline{u^*}]\Pi)}$$

- $\mathbf{R}^{v\downarrow}_{u\uparrow}$.

$$\dfrac{\dfrac{\mathcal{C} \to ([u\uparrow]\Gamma, [\underline{z}]\Lambda, [\underline{u^*}]\Pi)\ \text{ind. assumption}}{\mathcal{C} \to ([u\uparrow]\Gamma, [u\uparrow][\underline{z}]\Lambda, [\underline{u^*}]\Pi)}\ \mathbf{4}_{u,\underline{z}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \to [v\downarrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}{\mathcal{C} \to [v\downarrow\uparrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\ \mathbf{K}^{v\downarrow}_{v\downarrow\uparrow}}{\mathcal{C} \to [u\downarrow\uparrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\ \mathbf{K}^{v\downarrow\uparrow}_{u\downarrow\uparrow}}{\mathcal{C} \to [u\uparrow](p \leftarrow \Gamma, [\underline{z}]\Lambda)}\ \mathbf{K}^{u\downarrow\uparrow}_{u\uparrow}}{\mathcal{C} \to ([u\uparrow]p \leftarrow [u\uparrow]\Gamma, [u\uparrow][\underline{z}]\Lambda)}\ \mathbf{K}_{u\uparrow},\mathbf{C}_{u\uparrow}}{\mathcal{C} \to ([u\uparrow]p, [\underline{u^*}]\Pi)}$$

$\lhd$

4.9 Logics for Two Dimensions

We claimed in Section 4.3 that OO involves two evolutionary dimensions—object hierarchy and time—which gives a rise to a two-dimensional product hierarchy. In this section we show how from logics for two hierachies $\mathcal{O} = \langle O, \mathrm{isa}_O\rangle$, $\mathcal{T} = \langle T, \mathrm{isa}_T\rangle$, a logic for their product $\mathcal{U} = \langle U, \mathrm{isa}_U\rangle$ can be obtained. We present a general schema of how $2\mathrm{MU}_\mathcal{U}$, $2\mathrm{MU}'_\mathcal{U}$, and $2\mathrm{MU}''_\mathcal{U}$ emerge, and do not consider each logic separately in detail.

All the three logics $2\mathrm{MU}_\mathcal{U}$, $2\mathrm{MU}'_\mathcal{U}$, and $2\mathrm{MU}''_\mathcal{U}$ are normal multimodal logics, where the parameter set for the logic $2\mathrm{MU}_\mathcal{U}$ ($2\mathrm{MU}'_\mathcal{U}$, $2\mathrm{MU}''_\mathcal{U}$) is the union of the parameter sets of $2\mathrm{MU}_\mathcal{O}$ ($2\mathrm{MU}'_\mathcal{O}$, $2\mathrm{MU}''_\mathcal{O}$) and $2\mathrm{MU}_\mathcal{T}$ ($2\mathrm{MU}'_\mathcal{T}$, $2\mathrm{MU}''_\mathcal{T}$).

4.9.1 Motivation

Given an object $o \in O$ and a revision $\tau \in T$, we assume that the formulations "A in o at τ" and "A at τ in o" are equivalent. This predicts us that modalities with parameters from different one-dimensional logics should commute.

In Horn clauses, we see it necessary that there be two modalities over clauses, one of them being an object modality, and the other being a revision modality. Goals can be either non-modalized, or be covered with one or two modalities, where in the case of two modalities, again, one of them has to be an object modality, and the other a revision modality.

4.9.2 Axiomatics

The *Hilbert-style calculus* of $2\mathrm{MU}_\mathcal{U}$ ($2\mathrm{MU}'_\mathcal{U}$, $2\mathrm{MU}''_\mathcal{U}$) consists of the following rules and axioms:

- **The rules and axioms of the classical propositional Hilbert-style cal culus.**
- **Modal rules and axioms of** $2\mathrm{MU}_\mathcal{O}$ ($2\mathrm{MU}'_\mathcal{O}$, $2\mathrm{MU}''_\mathcal{O}$).
- **Modal rules and axioms of** $2\mathrm{MU}_\mathcal{T}$ ($2\mathrm{MU}'_\mathcal{T}$, $2\mathrm{MU}''_\mathcal{T}$).
- **Commutation axioms** For each parameter o of MU_O (MU'_O, MU''_O), and for each parameter τ of MU_T (MU'_T, MU''_T), we have:

$$[o][\tau]A \equiv [\tau][o]A,$$
$$[o]\langle\tau\rangle A \equiv \langle\tau\rangle[o]A.$$

4.9.3 Resolution calculus

The resolution calculi for $2\mathrm{MU}_{\mathcal{U}}$, $2\mathrm{MU}'_{\mathcal{U}}$ and $2\mathrm{MU}''_{\mathcal{U}}$can be obtained by combining the resolution calculi of $\mathrm{MU}_{\mathcal{O}}$ and $\mathrm{MU}_{\mathcal{T}}$, $\mathrm{MU}'_{\mathcal{O}}$ and $\mathrm{MU}'_{\mathcal{T}}$, and $\mathrm{MU}''_{\mathcal{O}}$ and $\mathrm{MU}''_{\mathcal{T}}$, respectively. In below, we present the resolution calculus for $2\mathrm{MU}_{\mathcal{U}}$.

Horn clauses are defined as follows for $2\mathrm{MU}_{\mathcal{U}}$:

For $2\mathrm{MU}_{\mathcal{U}}$, we define Horn clauses in the following way:

Definition 4.11 (Horn clauses) *Formulae $[v][\theta](p \leftarrow \Gamma, [\underline{z}]\Delta, [\underline{\vartheta}]\Sigma, [\underline{z'}][\underline{\vartheta'}]\Lambda)$, where p is an atomary formula, Γ, Δ, Σ and Λ are finite sets of atomary formulae, v is an object, θ is a revision, $\underline{z}$ and $\underline{z'}$ are finite sets of objects, and $\underline{\vartheta}$ and $\underline{\vartheta'}$ are finite sets of revisions, are called input clauses. Formulae $\leftarrow [\underline{u}][\underline{\tau}]\Pi$, where Π is a finite set of atomary formulae, $\underline{u}$ is a finite set of objects, and $\underline{u\tau}$ is a finite set of revisions, are called goal statements.*

The resolution calculus for $\mathrm{MU}_{\mathcal{U}}$ contains one rule:

- $\mathbf{R}^{(v,\theta)}_{(u,\tau)}$.

$$\frac{\leftarrow [u][\tau]p, [\underline{u^*}][\underline{\tau^*}]\Pi \qquad [v][\theta](p \leftarrow \Gamma, [\underline{z}]\Delta, [\underline{\vartheta}]\Sigma, [\underline{z'}][\underline{\vartheta'}]\Lambda)}{\leftarrow [u][\tau]\Gamma, [\underline{z}][\tau]\Delta, [u][\underline{\vartheta}]\Sigma, [\underline{z'}][\underline{\vartheta'}]\Lambda, [\underline{u^*}][\underline{\tau^*}]\Pi}$$

  if u $\mathrm{isa}^*_{\mathcal{O}}$ v and τ $\mathrm{isa}^*_{\mathcal{T}}$ θ.

4.10 Pragmatics

There are two pragmatical issues whose discussion we above postponed till this section: the overriding mode of inheritance, and representation of revisions (time instants). In the discussion of these, we use the $\mathrm{MU}'_{\mathcal{U}}$ notation for modalities.

4.10.1 Overriding

Though we did not provide any means to facilitate the overriding mode of inheritance in the designed logics, it still becomes possible to work with this mode on the pragmatical level, using cuts to this end.

In practical resolution, we assume that the goals in a goal statement are ordered, and that the clauses owned by same unit and defining same predicate are ordered. Given a predicate p and a unit u, the ordered set of u-owned defining clauses for p we call the definition of p at u. When resolving a goal p at u, we require that the definitions of p at u and its ancestors be attempted in accordance with their hierarchical order (bottom-up, left-to-right). Due to this convention, it is always easilily possible to rearrange the definition of p at a unit v, u isa* v, in such a way that the definitions of p at v's ancestors would never be attempted when resolving a goal at u. It is sufficient to add the following clause to the definition of p at v: (n is the arity of p)

```
[v↓]( p(X1,...,Xn) ← !, fail ).
```

Indeed, if resolution with the clauses from the definition of p at v failed, normally the definitions at v's ancestors would be tried. By the above addition, however, next the added clause will be attempted, the goal will always unify with the clause head successfully, and the result will be a non-backtrackable failure.

4.10.2 Revisions

The dimensions of object hierachy and time are not completely analogous. There are two significant characteristics of time instants (revisions) that objects do not have:

- Among the time instants, one is always distinguished—the *current time instant.* At any stage of evolution, this instant has no descendant instants. It is also usually (at least in case of linear time) assumed that new instants could be created only from this instant. In the object hierarchy, there is no correspondent to this phenomenon.
- When specifying in a program some action that would cause a new revision, we normally do not know, at which revision this action will be performed at run-time. Thus, clauses not only have to be able to relate explicitly given revisions, they also must be able to relate unspecified revisions about which only the action that causes transition from one to another, is known.

The idea of keeping trace after current instant seems to be alien to logic. Also, since the succession of actions is unknown before execution, the best we can do, is to allow all arbitrary sequences of actions and the resulting revisions of these sequences to be considered, and to leave it open which of these sequences take place in the actual time flow. We therefore identify revisions with sequences of actions, and define revisions and the isa relation between them as follows:

Definition 4.12 (Revisions, isa relation) *Assume we are given a set of actions Act. Then:*

1. *0 is revision;*
2. *if a is action and $\tau_1, \ldots, \tau_m$ are revisions, then $a(\tau_1, \ldots, \tau_m)$ is revision, and for any $i = \overline{1,m}$, $a(\tau_1, \ldots, \tau_m)$ isa τ_i.*

In such an approach, all the ancestors of a revision are contained in it as “suffixes”.

The described approach has strong commonality with using *streams* to model dynamics. The difference is that due to inheritance along the revisions hierachy, the frame problem has been solved. Indeed, if an action `act` has no influence on a predicate `p`, then in stream programming, we must write the following input clause:

```
p(act(Past), X1,...,Xn) ← p(Past, X1,...,X).
```

In our approach, goals

```
← [a(Past)]p(t1,...,tn)
```

are directly resolvable with suitable inheritable input clauses owned by `Past`, an the clause stating “no change” is unnecessary.

4.11 Examples

In this section, we provide two short examples. The first of them was used in Subsection 3.4.16, where we described Prolog++, and the second was used in Subsections 3.4.13 and 3.4.14, where we described ‘Objects as Intensions’.

4.11.1 Family Relationships Example

In this example, taken from [Mos90], the father of a family is stored only in the object for the eldest of children. For each person, his next older and next younger siblings are stored in his/her object. Data about the father for non-eldest children other siblings can be derived. Note that since we do have cumulation, we do not need to introduce the auxiliary property `eldestSibling` of `human` for computing fathers, which in Prolog++ was inevitable.

```
[human↓]( father(X) ← olderSibling(Y),[Y]father(X) )
[human↓]( married ← spouse(_) )
[human↓]( sibling(older,X) ← olderSibling(Y),[Y]sibling(older,X) )
[human↓]( sibling(older,X) ← olderSibling(X) )
[human↓]( sibling(younger,X) ← youngerSibling(X) )
[human↓]( sibling(younger,X) ← youngerSibling(Y),[Y]younger(older,X) )
[human↓]( brother(X) ← sibling(_,X),[X]sex(male) )
[human↓]( age(Age) ← [general]thisYear(Now),birth(Then), Age is Now-Then )

[male↓]( spouse(X) ← wife(X) )
[male↓]( retirementAge(65) )
[male↓]( sex(male) )

[chris]( name('Chris Moss') )
[chris]( height(183) )
[chris]( birth(1944) )
[chris]( wife(Karen) )
[chris]( olderSibling(jane) )
```

4.11.2 Stack Example

This example defines a standard stack which is empty at revision 0.

```
[stack][0] ( contents([]) )
[stack][0↓]( isempty ← contents([]) )
[stack][0↓]( top(El) ← contents([El|_]) )
[stack][push(El)(Past)↓]( contents([El|Rest]) ← !,[Past]contents(Rest) )
[stack][pop(El)(Past)↓]( contents(Rest) ← !,[Past]contents([El|Rest]) )
```

4.11.3 Evaluation

It can be seen from the simple examples which we provided here, that the designed logics have a rather good expressive power. They also demonstrate that despite the expressive power, the language variants are not overcomplicated and cause minimum overhead (recall that we wanted to have the internal layer general). We wish to underline once again that for the purpose of a concrete application area, it might be made a task of the concrete front-end layer to provide shorthands for syntactic constructs occurring more frequently than others.

For deeper evaluation, certainly, much bigger examples would be needed. Such evaluation, however, would require some working system prototype.

Chapter 5

Conclusions

The hope that it should be possible to exploit the merits of the OO and the logic paradigms simultaneously, has given an impetus to a search for a combination of those paradigms. Two of the problems that merger attempts have to face, are coping with modules and sharing (with both the implicit and explicit forms of access to objects) and coping with state changes. Our contribution to the study of these two problems can be summarized as follows.

We feel that in handling modules and sharing, the form of clauses that we adopted in the designed logics, is adequate w.r.t. the OO spirit. Indeed, we modalize each clause as a whole, and we also partially modalize the goals in its body. A clause modality indicates the object (or the object revision) to whose definition the clause belongs, and gives information on the accessibility of the clause by inheriters. Goal modalities point to the objects (object revisions) to whom the task of demonstrating these goals is appointed to and tell how the goals must be demonstrated. If a goal modality is omitted, then the clause modality is applied. Hence, essentially either the inheriter itself will have to demonstrate the goal, or a message is sent to some other object (object revision).

The simple but expressive clause forms shows that modal logic is a suitable tool for reasoning about modules and sharing.

It has also been one of the key ideas in the proposal of this thesis that evolution in OO is two-dimensional. We feel that it is really appropriate to consider the essential difference between attributes and methods to lie in that attribute values are inherited along the hierarchy of time instants, whereas methods are inherited along the object hierarchy. As we emphasized in Section 4.3, this solves the frame problem and also creates uniformity.

5.1 Future work

Though there has been much research going on during the recent years on possibilities to combine OO and logic, there are still many open questions in the field. Each of the mergers reported in literature solves some subproblems of the main problem, and so far none of them is universally satisfactory. There is therefore enough area in the realm for intensive further research.

With regards to the concrete proposal of the present thesis, the following steps should be taken to reach at a prototype of a working system:

- Detailed design of the front-end layer.

- Implementation.

Also, the following directions should be worthwhile of future study:

- Kripke semantics for $\mathrm{MU}''_{\mathcal{U}}$. Alternative approaches to build up $\mathrm{MU}'_{\mathcal{U}}$ and $\mathrm{MU}''_{\mathcal{U}}$, where the only parameters are units and symbols $\downarrow$, $\uparrow$.
- Possible link between the overriding mode of inheritance and linear logic.
- Other solutions than the present for coping with revisions.
- Dynamic object hierarchies. At the present stage, the designed logics allow only objects be dynamic, while the isa links between them must be static.

Only when the present thesis was in the stage of completion, the author got to know about the work by M. Kifer et al. [KLW90]. Even basing on superficial acquaintance with the mentioned work, it is evident that a major breaktrough has been made there. The connection between the formalisms of the present thesis and that work deserves studying.

There is also an obvious necessity to consider some larger examples in order to find out what lessons could be learnt from these.

Bibliography

[AM86] M. Abadi and Z. Manna. Modal theorem proving. In J. H. Siekmann, editor, *8th Int'l Conf. on Automated Deduction: Proc., Oxford, UK, 27 July – 1 Aug 1986*, pages 172–89. Springer, Berlin-Verlag, 1986. (*LNCS*, 230).

[AN86] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[BGHS91] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, editors. *Object-Oriented Languages, Systems and Applications.* Pitman, London, 1991. 375 pp.

[BK82] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 153–72. Academic Press, London, 1982.

[BMS90] M. v. Biema, G. Q. Maguire, and S. Stolfo. The constraint-based paradigm: Integrating object-oriented and rule-based programming. In *Proc. 23rd Annual Hawaii Int'l Conf. on Syst. Sci., Kailua-Kona, HI, USA, 2–5 Jan 1990*, volume 2, pages 358–66. IEEE Comp. Soc. Press, Los Alamitos, CA, 1990.

[BS86] D. G. Bobrow and M. J. Stefik. Perspectives on artificial intelligence programming. *Science*, 231:951–7, 1986.

[Cat88] L. Catach. Normal multimodal logics. In *AAAI-88: Proc. 7th Nat'l Conf. on Artif. Intelligence, St. Paul, MI, USA, 21–26 Aug 1988*, volume 2, pages 491–5. 1988.

[Cha87] M. Chan. The recursive resolution method for modal logic. *New. Gener. Computing*, 5(1):155–183, 1987.

[Che80] B. F. Chellas. *Modal Logic: An Introduction.* University Press, Cambridge, UK, 1980. 295 pp.

[Che87] W. Chen. A theory of modules based on second-order logic. In *Proc. 1987 Symp. on Logic Programming, San Fransisco, CA, USA, 31 Aug – 4 Sep 1987*, pages 24–33. IEEE Comp. Soc. Press, Washington, DC, 1987.

[Chi84] T. Chikayama. Unique features of ESP. In *Proc. 1984 Int'l Conf. on Fifth Gener. Comput. Syst., Tokyo, Japan, 6–9 Nov 1984*, pages 292–8. North-Holland, Amsterdam, 1984.

[Con88] J. S. Conery. Logical objects. In *Logic Programming: Proc. 5th Int'l Conf. and Symp., Seattle, WA, USA, 15–19 Aug 1988*, pages 420–34. The MIT Press, Cambridge, MA, 1988.

[CW88] W. Chen and D. S. Warren. Objects as intensions. In *Logic Programming: Proc. 5th Int'l Conf. and Symp., Seattle, WA, USA, 15–19 Aug 1988*, pages 404–19. The MIT Press, Cambridge, MA, 1988.

[Day89] G. Dayantis. Types, modularization and abstraction in logic programming. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Algebraic and Logic Programming: Int'l Workshop: Proc., Gaussig, GDR, 14–18 Nov 1988*, pages 127–36. Springer-Verlag, Berlin, 1989. (*LNCS*, 343).

[DH91] R. Ducornau and M. Habib. Masking and conflicts, or to inherit is not to own. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 223–44. J. Wiley & Sons, Chichester, UK, 1991.

[Far86] L. Fariñas del Cerro. MOLOG: A system that extends Prolog with modal logic. *New Gener. Computing*, 4(1):35–50, 1986.

[FH86] K. Fukunaga and S. Hirose. An experience with a Prolog-based object-oriented language. In N. Meyrowitz, editor, *OOPSLA'86: Object-Oriented Programming Systems, Languages and Applications: Conf. Proc., Portland, OR, USA, 29 Sep – 2 Oct 1986*, pages 224–31. 1986. (*SIGPLAN Notices*, 21(11)).

[FTK+84] K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: A logic based knowledge programming system. In *Proc. 1984 Int'l Conf. on Fifth Gener. Comput. Syst., Tokyo, Japan, 6–9 Nov 1984*, pages 613–22. North-Holland, Amsterdam, 1984.

[FY88] Chen Fu-an and Zhu Yi-fen. POKRS: A Prolog-based object-oriented knowledge representation system. In *Proc. 1988 Int'l Conf. on Syst., Man and Cybern., Beijing and Shenyang, China, 8–12 Aug 1988*, volume 1, pages 285–8. Int'l Acad. Publ., Beijing, 1988.

[Gal86] H. Gallaire. Merging objects and logic programming: Relational semantics. In *AAAI-86: Proc. 5th Nat'l Conf. on AI, Philadelfia, PA, USA, 11–15 Aug 1986*, volume 2, pages 754–8. Morgan Kaufmann, Los Altos, CA, 1986.

[GM87] J. A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–77. The MIT Press, Cambridge, MA, 1987.

[HE90] B. Henderson-Sellers and J. M. Edwards. Object-oriented systems life cycle. *Comm. ACM*, 33(9):142–59, 1990.

[HV87] M. Huber and I. Varsek. Extended Prolog for order-sorted resolution. In *Proc. 1987 Symp. on Logic Programming, San Fransisco, CA, USA, 31 Aug – 4 Sep 1987*, pages 34–43. IEEE Comp. Soc. Press, Washington, DC, 1987.

[IC90] M. H. Ibrahim and F. A. Cummins. KSL/Logic: Integration of logic with objects. In *Proc. 1990 Int'l Conf. on Comp. Languages, New Orleans, LA, USA, 12–15 Mar 1990*, pages 228–35. IEEE Comp. Soc. Press, Los Alamitos, CA, 1990.

[IT86] Y. Ishikawa and M. Tokoro. Concurrent object-oriented knowledge representation language Orient84/K: Its features and implementation. In N. Meyrowitz, editor, *OOPSLA'86: Object-Oriented Programming Systems, Languages and Applications: Conf. Proc., Portland, OR, USA, 29 Sep – 2 Oct 1986*, pages 232–41. 1986. (*SIGPLAN Notices*, 21(11)).

[JL88] J. Jaffar and J.-L. Lassez. From unification to constraints. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '87: Proc. 6th Conf., Tokyo, Japan, 22–24 June 1987*. Springer-Verlag, Berlin, 1988. (*LNCS*, 315).

[KE88] T. Koschmann and M. W. Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 5(5):36–42, 1988.

[KG86] H. Kauffmann and A. Grumbach. MULTILOG: MULtiple worlds in LOGic programming. In *ECAI'86: 7th Europ. Conf. on Artif. Intelligence: Proc., Brighton, UK, 21–25 July 1986*, volume 1, pages 291–305. 1986.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14, SUNY at Stony Brook, Dept. of Comput. Sci., August 1990. 84 pp.

[KM90] T. Korson and J. D. McGregor. Understanding object-oriented: A unifying paradigm. *Comm. ACM*, 33(9):40–60, 1990.

[KTMB87] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Vulcan: Logical concurrent objects. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 75–112. The MIT Press, Cambridge, MA, 1987.

[LM88] L. Leonardi and P. Mello. Combining logic- and object-oriented programming language paradigms. In B. D. Shriver, editor, *Proc. 21st Annual Hawaii Int'l Conf. on Syst. Sci.*, volume 2, pages 376–85. IEEE Comp. Soc. Press, Washington, DC, 1988.

[LV91] E. Laenens and D. Vermeir. A logical basis for object-oriented programming. In *Logics in AI: European Workshop JELIA'90: Proc., Amsterdam, NL, 10–14 Sep 1990*, pages 317–32. Springer-Verlag, Berlin, 1991. (*LNAI*, 478).

[LVV89] E. Laenens, D. Vermeir, and B. Verdonk. LOCO, a logic-based language for complex objects. In *ESPRIT'89: Proc. 6th Annual ESPRIT Conf., Brussels, Belgium, 27 Nov – 1 Dec 1989*, pages 604–16. Kluwer Acad. Publ., Dordrecht et al., 1989.

[Mel88] F. Mellender. An integration of logic and object-oriented programming. *SIGPLAN Notices*, 23(10):181–5, 1988.

[Mel91] P. Mello. Inheritance as combination of Horn clause theories. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 275–89. J. Wiley & Sons, Chichester, UK, 1991.

[Mes90] J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proc.: Conf. on Object-Oriented Programming: Systems, Languages and Applications, Europ. Conf. on Object-Oriented Programming, Ottawa, Canada, 21–25 Oct 1990*, pages 101–15. 1990. (*SIGPLAN Notices*, 20(10)).

[Mil86] D. Miller. A theory of modules for logic programming. In *Proc. 1986 Symp. on Logic Programming, Salt Lake City, UT, USA, 22–25 Sep 1986*, pages 106–14. IEEE Comp. Soc. Press, Washington, DC, 1986.

[MLV89] J. Malenfant, G. Lapalme, and J. Vaucher. ObjVProlog: Metaclasses in logic. In S. Cook, editor, *ECOOP'89: Proc. 1989 Europ. Conf. on Object-Oriented Programming, Nottingham, UK, 10–14 July 1989*, pages 257–69. Univ. Press, Cambridge, UK, 1989.

[MN86] P. Mello and A. Natali. Programs as collections of communicating Prolog units. In B. Robinet and R. Wilhelm, editors, *ESOP 86: Europ. Symp. on Programming: Proc., Saarbrücken, Germany, 17–19 Mar 1986*, pages 274–88. Springer-Verlag, Berlin, 1986. (*LNCS*, 213).

[MN87] P. Mello and A. Natali. Objects as communicating Prolog units. In J. Bezivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP'87: Europ. Conf. on Object-Oriented Programming: Proc., Paris, France, 15–17 June 1987*, pages 181–92. Springer-Verlag, Berlin, 1987. (*LNCS*, 276).

[MOK84] F. Misoguchi, H. Owhada, and Y. Katayama. LOOKS: Knowledge representation system for designing expert systems in logic programming framework. In *Proc. 1984 Int'l Conf. on Fifth Gener. Comput. Syst., Tokyo, Japan, 6–9 Nov 1984*, pages 606–12. North-Holland, Amsterdam, 1984.

[Mos90] C. Moss. An introduction to Prolog++. Research Report DOC 90/10, Imperial College, London, June 1990.

[MP90] L. Monteiro and A. Porto. Semantic and syntactic inheritance in logic programming. Draft report, Departamento de Informatica, Universidade Nova de Lisboa, December 1990. 12 pp.

[MT90] G. E. Mints and E. H. Tyugu. Propositional logic programming and the PRIZ system. *Journal of Logic Programming*, 9(3):179–93, 1990.

[Nak84] H. Nakashima. Knowledge representation in Prolog/KR. In *1984 Int'l Symp. on Logic Programming, Atlantic City, NJ, USA, 6–9 Feb 1984: Conf. Proc.*, pages 126–30. IEEE Comp. Soc. Press, Silver Spring, MD, 1984.

[Obe89] A. Oberschelp. Order sorted predicate logic. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artif. Intelligence: Workshop Proc., Eringerfeld, FRG, 24–26 Apr 1989*, pages 8–17. Springer-Verlag, Berlin, 1989. (*LNAI*, 418).

[Ohl88] H. J. Ohlbach. A resolution calculus for modal logics. In E. Lusk and R. Overbeek, editors, *9th Int'l Conf. on Automated Deduction: Proc., Argonne, IL, USA, 23–26 May 1988*, pages 500–16. Springer-Verlag, Berlin, 1988. (*LNCS*, 310).

[Ohl90] H. J. Ohlbach. New ways for developing proof theories for first-order multimodal logics. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89: 3rd Workshop on Comp. Sci. Logic: Proc., Kaiserslautern, FRG, 2–6 Oct 1989*, pages 271–308. Springer-Verlag, Berlin, 1990. (*LNCS*, 440).

[Sch89] P. H. Schmidt. Tableau calculus for order sorted logic. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artif. Intelligence: Workshop Proc., Eringerfeld, FRG, 24–26 Apr 1989*, pages 49–60. Springer-Verlag, Berlin, 1989. (*LNAI*, 418).

[Sho88] Y. Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* The MIT Press, Cambridge, MA, 1988. 201 pp.

[ST83] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Gener. Computing*, 1(1):25–48, 1983.

[Ste87] L. A. Stein. Delegation is inheritance. In N. Meyrowitz, editor, *OOPSLA'87: Object-Oriented Programming Systems, Languages and Applications: Conf.*

Proc., Orlando, FL, USA, 4–8 Oct 1987, pages 138–46. 1987. (*SIGPLAN Notices*, 22(12)).

[Str88] B. Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, 1988.

[TI84] M. Tokoro and Y. Ishikawa. An object-oriented approach to knowledge systems. In *Proc. 1984 Int'l Conf. on Fifth Gener. Comput. Syst., Tokyo, Japan, 6–9 Nov 1984*, pages 623–31. North-Holland, Amsterdam, 1984.

[VLM88] J. Vaucher, G. Lapalme, and J. Malenfant. SCOOP: Structured concurrent object-oriented Prolog. In S. Gjessing and K. Nygaard, editors, *ECOOP'88: Europ. Conf. on Object-Oriented Programming: Proc., Oslo, Norway, 15–17 Aug 1988*, pages 191–211. Springer-Verlag, Berlin, 1988. (*LNCS*, 322).

[Wal88] L. A. Wallen. Automated theorem proving in non-classical logics. In T. Danielsen, editor, *Scand. Conf. on Artif. Intelligence: Proc. SCAI'88, Tromsø, Norway, 9–11 Mar 1988*, pages 1–12. IOS, Amsterdam, 1988.

[Wal89] C. Walther. Many-sorted inferences in automated theorem proving. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artif. Intelligence: Workshop Proc., Eringerfeld, FRG, 24–26 Apr 1989*, pages 19–48. Springer-Verlag, Berlin, 1989. (*LNAI*, 418).

[Weg87] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. The MIT Press, Cambridge, MA, 1987.

[WWE89] S. N. Woodfield, J. M. Weston, and D. W. Embley. Combining three conceptual models: Abstract data types, logic programming, and databases. In *8th Annual Int'l Phoenix Conf. on Computers and Comm's: 1989 Conf. Proc., Scottsdale, AZ, USA, 22–24 Mar 1989*, pages 322–6. IEEE Comp. Soc. Press, Washington, DC, 1989.

[Zan84] C. Zaniolo. Object-oriented programming in Prolog. In *Proc. 1984 Int'l Symp. on Logic. Programming, Atlantic City, NJ, USA, 6–9 Feb 1984*, pages 265–70. IEEE Comp. Soc. Press, Silver Spring, MD, 1984.