From Ruby to Rudy

First steps...to Diamondize Ruby

© by Rudolf Kaehr
ThinkArt Lab Glasgow January 2006

DERRIDA'S MACHINES PART III

BYTES & PIECES

of

PolyLogics, m-Lambda Calculi, ConTeXtures

From Ruby to Rudy

First steps...to Diamondize Ruby

© by Rudolf Kaehr

ThinkArt Lab Glasgow January 2006

"Interactivity is all there is to write about:
it is the paradox and
the horizon of realization."

From Ruby to Rudy

First steps...to Diamondize Ruby

Some Motivational Remarks

From Ruby to Rudy

First steps...to Diamondize Ruby

Time and Computation

Some Motivational Remarks

1 Contextural Programming

1.1 What is Contextural Programming?

Essentials of classic programming

"A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. [...]
Every powerful language has three mechanisms for accomplishing this:
primitive expressions, which represent the simplest entities with which the language is concerned,
means of combination, by which compound expressions are built from simpler ones, and
means of abstraction, by which compound objects can be named and manipulated as units.
In programming, we deal with two kinds of objects: procedures and data." Sussman, p. 4

In terms of the lambda based proto-language ARS:

"*Abstraction:* Give something a name.
Reference: Reference an abstraction by name.
Synthesis: Combine two or more abstractions to create a new abstraction." Loczewski

What do we understand by *"contextural programming"?*

Programming in the mono-contextural sense, that is, programming as we know it, is characterized by the linguistic operations of abstraction, reference and synthesis as Loczewski puts it in the tradition of Abelson/Sussman.

What is not mentioned in the characterization of classic programming is the uniqueness of the programming system. There is, conceptually, one and only one programming language. As there is one and only one logic, arithmetic, mathematics. This not contradicting the 10000+1 different programming languages on the growing market. They are all introduced as being singular, conceived in the framework of universal computability. Computation, organized by such a conception of programming languages, is, in a general sense, information processing or message passing.

In contrast, polycontextural programming starts, as the name intends, with a plurality of contextures. Each contexture is giving place to the realization of a located programming language and its conception of computability.

Thus, the first observation is the fact that classic programming is realized inside of one and only one contexture. As a consequence classic programming languages are blind to this fact and are not able to discover their computational environment. This can be called the *Blind Spot* of mono-contextural programming. With the fact of the Blind Spot, *reflectionality* and *interactivity* are excluded for systematic reasons.

Multitudes in classic programming languages occur as the multitude of programming *styles* (paradigms) and *topics* (List, Boolean, Numeric, etc.) and their methods of manipulations.

Because it is involved in the activity of *thematization*, in contrast to *abstraction*, the strategy of contextures is always engaging a multitude of disseminated contextures, i.e, *poly-contexturality* in contrast to *mono-contexturality*.

Contextural programming is designing the dissemination of contextures.

Computation, organized by such a conception of contextural programming, is basically not concerned with information processing or message passing. But with the mechanisms of *togetherness* of contextures which is realized as *interactionality* and *reflectionality*, but also with *interventionality* and *interlocutionality*. Traditional terms like "interaction" or "reflection" are used for message passing, say in MAS, and are thus misleading in a polycontextural framework. If used, then they are understood only as linguistic abbreviations of the contextural terms.

Togetherness of contextures is emphasizing the *loci* or positions (positionality) contextures are occupying/enabling in the graphematic game of inscriptions. Only located contextures can be mediated and distributed. Locatedness and togetherness of contextures are enabling the mechanisms of dissemination, i.e., distribution and mediation as polycontextural procedures. Locatedness and togetherness, as such, are characterized and involved in the general theory and mechanism of loci, i.e., of morpho- and kenogrammatics.

In other words, togetherness is a characteristics of living systems. The main criteria of contextural programming, therefore, are not derivability, truth and computation but togetherness and liveliness of co-existing systems.

What is meant by *contextures*?

The main concept in ConTeXtures obviously is "contexture". It could be said that a contexture is the context of all possible contexts of a language. Like a universal domain in first order logic a contexture has no limits. With this, we are in the middle of paradoxes and antinomies. Because what we call the context of all possible contexts, a single contexture, is only one of the plurality of contextures as it is defined by polycontexturality. It even makes not much sense to speak about a single contexture. In other words, the ultimate generality of contexture is given by the universal application of the Excluded Middle (TND). Classical thinking happens in the framework of one contexturality, thus it is called mono-contextural. But it is christened with mono-contexturality only from the outside of it, that is, from the position of polycontexturality. On the other hand, a single contexture is not a simple unity, a basic contexture like an atom, but stands relative to a compound contexture consisting of a plurality of contextures. As Gunther puts it: *"It cannot be too strongly emphasized that the distinction between elementary contexture and compound contexture is relative."*

Contextural programming, therefore, is not based on "primitive expressions". Contextures are not the atomic elements of a referencing action, like in classic programming, but ambigous "building-blocks" of dynamic complexity, interchanging between elementarity and compoundedness. Primitive expressions are involved in linear and hierarchic orders, contextures are entangled in a heterarchic and tabular game guided by the proemial relation (chiasm).

Dissemination of contextures is realized as the complementarity of distribution and mediation of contextures.

The kind of distribution is guided by the architectonics of the situation to be thematized for programming. The architectonics of programming systems are defined by *complexity* and *complication* of the design and their *linear*, *tabular* or *circular* organization.

Classic programming paradigms are based on abstractions. Transclassic programming paradigms are based on *thematizations*. Thematizations are complex contextural abstractions which are placing at each contexture classic, say lambda, abstractions.

Interpretations of architectonics and scenarios

On a first level of introduction, contextural programming can be considered as the programming of the complementary aspects of *reflectionality* and *interactionality* of disseminated contextures as a specific realization of the complexity/complication of the architectonics. Higher designs of complexity/complication can be interpreted as *intervention* and *interlocution* of computational systems.

Therefore, contextural programming is considered with the computational rules and laws between disseminated contextures. Each contexture is giving space for a full classic programming language. Contextural programming is developing the inter-computational situations of disseminated programming languages of different styles and topics.

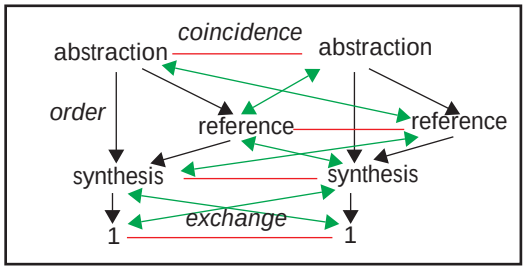
To concretize the dissemination of different programming languages a step by step mediation of their distributed components has to be done. Within the framework of disseminated contextures, based on a chosen architectonics, implying complexity and complication of the polycontextural compound, all the conceptual components of the different disseminated programming paradigms have to be mediated on the their proper systematic levels.

Mediation pattern: Structure + Dynamics = Structuration

First, *structure*: All three main mechanisms of the distributed languages, which are structured by the order relations, have to be mediated in the mode of the *non-metamorphic as-abstraction*, i.e., in a one-to-one correspondence in the sense of the coincidence relations of proemiality:

1. the mechanisms of abstraction as abstraction,
2. the mechanisms of reference as reference, and
3. the mechanisms of synthesis as synthesis.

Second, *dynamics*: All three main mechanisms of the distributed languages have to be involved into the creative game of *metamorphic as-abstractions*, i.e., thematizations, including exchange relations:



4. Abstraction as reference,
5. Abstraction as synthesis,
6. Referencing as abstraction,
7. Referencing as synthesis,
8. Synthesis as abstraction, and
9. Synthesis as reference.

This mediation pattern of the main mechanisms of programming can itself be iterated, depending on the complexity/complication of the architectonics of the programming system.

The mediation pattern can be considered as a second-order mechanism of programming language construction: the language of the language, i.e., ARS(ARS).

Then, polycontextural programming is: DISS(ARS)$^{(m, n)}$.

The first mechanism of dissemination is constituting the *structure* of a complexion.The second is constituting the *dynamics* of complex programming systems. Both together, as a complementarity, emphasizing the dynamics of structures and the structure of dynamics, are called *structuration*.

1.2 Shifts in terminology: From instantiation to dissemination

Abstraction vs. thematization

Abstraction is the process of identifying compound expressions or objects as units. Thus, abstraction is naming identified objects by giving them a unique name. Abstractions as identifiers are realized by is-abstractions in contrast to as-abstractions of thematizations. Is-abstractions are working in the linguistic model of sentences and propositions and can be repeated at each contextural locus, while as-abstractions are realized in the medium of inter-textuality.

As a generalization of the conceptual abstraction we have to develop a shift to the textual mechanism of thematization. Linguistically, thematizing is not referring but evoking. Evocation (lit. Webster, 'imaginative recreation') is a hermeneutic process, thus creative and risky, not identifying but interpreting textures. Interpretations are depending on view-points, represented as contextures, and their interlocking mechanisms.

Thematizing, as an *epistemic activity* ("abstraction") of thematizing the world happens as plurality; there are always a multitude of viewpoints of thematizing. These viewpoints which opens up contextures are not isolated events, they are interacting and being together, they are mediated and the rules how to move from one point of view to another have to be given. Thematization as interpretation and/or thematization as identification. Identification, again is, *"giving something a name"*, that is, identification is abstraction, abstracting identity, an identical property, out of complexity and diversity. Abstraction as identification is the sense of and behind the lambda calculus. To identify is to iterate the same as the identical. And this kind of identification determines the kind of iterability of the operations.

What is *abstraction* for the lambda calculus is *identification* for combinatory logic.

Reference vs. evocation

Referencing something is to select it out of other entities. Such referenced entities are belonging to their corresponding contexture and are basically "primitive expressions", instances, as elements of further combinations (synthesis) of the language.

Contextural programming is electing, i.e. evocating contextures and is not primarily selecting pre-given atomic terms. Contextures are not the atomic elements of a referencing action but imaginative and ambivalent "building-blocks" of dynamic complexity, interchanging between elementarity and compoundedness. Primitive expressions are involved in linear and hierarchic orders, thus founding selection, contextures are entangled in a heterarchic and tabular game, thus founding election. Primitive expressions are involved in synthesis, contextures are entangled in dissemination

Synthesis vs. mediation

Synthesis is building compound expressions out of simple expressions. The structure of such a synthesis is a linearly ordered concatenation of atomic elements. The components of synthesis are happening in the framework of an established contexture. Synthesis is an intra-contextural operation, while mediation is a trans- and inter-contextural mechanism which is operating between distributed contextures. In polycontextural systems, mediation and distribution are complementary terms.

To put both activities together, the term *dissemination* (Derrida) is used. But dissemination is not a simple term of combining two other elementary terms, but an irreducibly ambivalent and antinomic non-term. Like contextures, dissemination is not given or perceptibly/thinkable in a single act of evidence (Husserl). The antinomy is created by the simultaneous dissimilarity of the distributed sameness of contextures. *"Dissemination 'is' (about) the play of meanings; an unequivocal meaning cannot be assigned to it."*

> "Dissemination 'is' a scattering of semen, seeds and semes, semantic features. 'We are playing on the fortuitous resemblance ... of seme and semen. There is no communication of meaning between them. And yet, by this floating, purely exterior collusion, accident produces a kind of semantic mirage: the deviance of meaning, its reflection-effect in writing, sets something off ... it is a question of remarking a nerve, a fold, an angle that interrupts totality: in a certain place, a place of well-determined form, no series of semantic valences can any longer be closed or reassembled ... the lack and the surplus can never be stabilized in the plenitude of a form' "(Derrida, Positions, p.45-6).

Position vs. distribution

The positioning of a formal system or programming language is produced by the decision of its introduction. The language with its introduced definitions and features is positioned by uniqueness. As a formal system a programming language is defining its uniqueness and positionality by its meta-theoretical characteristics. In fact, it fails to do so because it is not able to motivate the motivation of the decision. There is no working self-definition or self-motivation of a formal language (Gödel). Motivations for the decision toward the positioning of the formal systems are not parts of the systems. They are outside, somewhere in the tradition and ruled by specific interests of the historic formation of technological culture.

Distribution of contextures is designed by the architectonics of the mediated formal languages. In this sense, classical positioning of a language is a singular distribution, producing a punctual one-element architectonics.

In analogy to the positional system of number theory it could be said that contextures are distributed over a complex topology of different positions, loci, of a tabular notational system.

1.3 Poly-paradigm of contextural programming

Uni-paradigm

Often the definition or choice of a paradigm is related to a privileged data type out of possible topics. But this is a mere coincidence between a kind of abstraction and the privileged topic. Paradigms are based on the basic abstraction of the programming language and are supported by the way it is realized. Thus, paradigms are a kind of a style of programming. Depending on the tasks, styles can be changed and a couple of styles, applied together is introducing the multi-paradigm approach.

Multi-paradigms

Multi-paradigm programming is based on a general abstraction and a introduction of the programming language which is broad enough to allow different programming styles and therefore different decisions which data type and style is privileged to other implicit privileges and decisions of styles. But each decision is singular at its systematic place. The multi-paradigm approach allows to change paradigm to solve different tasks in a save way. This kind of switching from one paradigm to another is sequential and is not realized in a parallel or concurrent manner at a systematic place of a concern or task. Different paradigms in multi-paradigm programming exist in separation and are not mediated together to build a complexion, because their concerns, tasks, objects can be treated in a sequential way, without the need of any simultaneity.

Poly-paradigms

Poly-paradigm programming takes into account the possibility of a simultaneous plurality of different programming styles. Thus, an object can be thematized at once as belonging to different topics and styles. Such an computational object thus is not a singularity but a dynamic complexion.

Uni-paradigm and multi-paradigm approaches are based in a common singular programming framework. Both are therefore mono-contextural. The poly-paradigm approach is essentially involved in the plurality of different disseminated frameworks of programming, thus irreducibly polycontextural. It is based on the polycontexturality of complex programming and not on its styles or topics.

1.4 Poly-topics in contextural programming

Poly-topics are disseminated topics. They are specifications of the general complex objects, *c-obs*. Topics are specified as the data types or sorts, like Boolean, Numeric, List, Relational, Class, etc. and reflectional/interactional realizations. Replications of the same topics in different contextures are called *mono-form* topics. Otherwise *poly-form* topics. Topics are not identities per se, they can be involved in transformations by the process of interactional/reflectional *metamorphic* abstractions.

1.5 Main operational mechanisms between contextures

Between disseminated contextures of a contextural complexion (compound) different transformational mechanisms can be introduced.

The main transformations between contextures are collected by the set of *super-operators*, sops = {id: identity, perm: permutation, red: reduction, repl: replication, bif: bifurcation}. An operator *spec* for *specify* is introduced, additionally to the general operation *identify*, for reflectional replication of contextures.

Super-operators may play two roles. One as transforming operators and one as meta-instructions. Meta-instructions play a similar role as *define* or *lambda* in the distributed calculi. Thus, meta-instructions are introduced as the operators of thematization *thematize,* collected as {identify, permute, replicate, reduce, bifurcate, specify}.

2 Why should we need contextural programming?

"It should now be understood if we say that the classic, two-valued logic describes our system of formal rationality as an undistributed order of concepts. This is done by vigorously excluding any reference to the thinking subject.

To sum it up: *A non-Aristotelian or trans-classical logic is a system of distributed rationality. Our traditional logic presents human rationality in a non-distributed form. This means: the tradition recognizes only one single universal subject as the carrier of logical operations. A non-Aristotelian logic, however, takes into account the fact that subjectivity is ontologically distributed over a plurality of subject-centres. And since each of them is entitled to be the subject of logic human rationality must also be represented in a distributed form."*

Gotthard Gunther, The Tradition of Logic and The Concept af a Trans-classical Rationality. http://www.vordenker.de/ggphilosophy/gg_tradition-of-logic.pdf

Even if there is no need at all for contextural programming and for polycontexturality in general, the option to discover it has been opened up to the textu(r)al play. War games have engineered programming to great historic importance. The proposed paradigm is developing first steps of programming as the joy of the cosmic life-game.

2.1 A general framework for distributed rationality

Agents and multi-agent systems in *ConTeXtures* are studying the *architectonic* aspect of multi-agent systems as the conditions of interactional, reflectional, environmental and computational behaviors of agents including the structure of Interaction, Organization and Environment of Agents in multi-agent systems (MAS).

To adapt a more familiar wording I will try to put, in a very first step, the designed polycontextural concepts in a framework similar to the well known theories of agents, like MAS (Pfalzgraf, Wooldrige) and MIC (Gouaich, Zambonelli).

"According to Demazeau [Dem95] a multi-agent system can be described with four main concepts: Agent, Interaction, Organisation and Environment*."* (Zambonelli)

An *agent* in ConTeXtures is a subject (actor, entity) with an environment realizing that in its environment there are other subjects with their own environments containing himself in their environment. Thus, an agent as a subject is both at once an agent and part of an environment of an agent. An agent *has* an environment and *is part* of an environment. With this, a theory of agents in ConTeXtures starts conceptually not with one but with at least two agents in the game. This duplicity of agents is not defined by superposition of actors but as architectonic simultaneity of interacting agents.

Architectonics

Architectonics is prior to the quadruple of (agent, interaction, organization, environment) and is attempting to conceptualize the structure of *togetherness* of subjects in societal systems. The architectonic conditions for agents building a societal system is not yet guaranteeing successful communication, co-operation, interaction, etc. on an informational level but is conceived as its pre-conditions.

Further more, an agent has an *inner* and a *outer* environment. In his outer environment he confronts other agents and an environment neutral to agents. In his inner environment he reflects the outer environment in its two ways as neutral and as actional. He also reflects in his inner environment his own behavior to his environment, especially to his interactional environment, that is, to other agents and their behaviors.

Coalitions of agents to societal systems are architectonically *super-additive*. The cooperation of two agents is producing a new interactional space (contexture) which is modeling the difference and mediation of the two acting agents as the realization of cooperation. Two co-operating agents are realizing cooperation as a third actional space (object, contexture).

Modi of actions

The aim of *interaction* is to change the inner environment of an agent by changing its outer environment. Thus, *interaction* is a change of model at the place of the neighbor agent.

The aim of *reflection* is to change the inner environment of the reflecting agent itself. (Self-reflection, Introspection, Memory) Thus, *reflection* is a change of model at the place of the agent itself. This change is purely structural and neutral to specific informations.

The aim of *intervention* is to change the inner environment of an agent by changing its behavioral environment. (Coaching, Consultation, Re-programming)

The aim of *anticipation* is to change the inner environment of an agent by confronting him with the inner environment of the acting agent. (Motivations, Desires, Emotions)

Rationality and Super-additivity

Each agent has its *rationality* and *computability*. The rationality of multi-agent systems is super-additive. Rationality of an agent can be modeled by its logic. Thus, combining logics for MAS is super-additive. Topics of super-additivity don't exist in MAS and Combining Logics. Exceptions can be found in the work of Pfalzgraf and Beziou.

Movements in ConTeXture are defined as the processes of coalition building by different agents resulting in a societal and super-additive compound structure.

Questions of *time* and *space* of activities of agents in the sense of *temporal* and *spacial* attributes enters secondary into the game of interactionality / reflectionality of agents.

Actions of agents in *societal architectonics* take place, are situated and thus, agents are occupying a structural place, a *locus,* for their realization.

Inner and outer environments are mirrored, represented by *models*, realized by the agents, occupying a place in the actional space (contexture) of an agent. The actional space of an agent is pre-spacial and pre-temporal.

Togetherness / strangeness, acceptance / rejection are the main categories or rationals of a theory of interacting agents.

Proemiality of interaction and reflection

The general matrix of interactivity and reflectionality is not simply be pre-given but has to be established. The rules of the preconditions of the matrix are defined by the proemial relation ship. Also the disolvement of togetherness is described by the rules of proemiality.

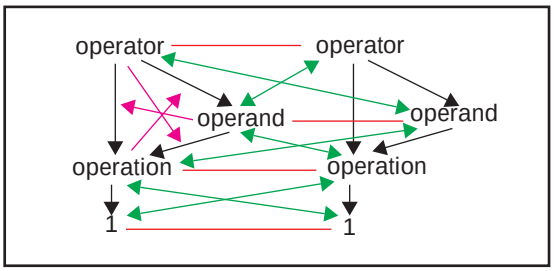
To be able to realize a concrete interaction the proemiality or chiasm of inside / outside of at least two agents has to be fulfilled. Second to the exchange relations of inside / outside in respect of agent1 and agent2 a coincidence or correspondence relation has to be realized. The correspondence which has to be realized is the correspondence between the model of agent1 of agent2 and the model of agent2 of agent1. The coincidence relation has to guarantee that there is a correspondence between inner models and not between something else. The correspondence has to be from model to model. A second correspondence has to be realized in a similar way between agent1 and agent2, that is, between agents and not something else. There is no correspondence between, say, an agent and an inner model of an agent. Between agent and model an exchange relation has to be realized.

And, obviously, for each agent the existence of an order relation between inside and outside, that is the inner model of an agent and an agent as an entity of an outer environment, has to be established.

Blind Spot Problem and the Otherness of the Others

Non-distributed rationality, i.e., mono-contexturality in logics and programming paradigms, is producing at least two crucial shortfalls: lack of architectonic *reflectionality* and *interactionality* as consequences of the structural inability of classic thinking to recognize the *Blind Spot Problem* of its mono-contexturality and to accept the *Otherness of the Other*. Excluding both, architectonic reflectionality and interactionality, there is no access to an understanding of systemic *intervention* and anticipative *interlocution*.

2.2 Blind Spot Problem: No reflectionality



Reflectionality comes into the general game if we thematize the relationality or operativity of the proemial construction from the point of view of an *internal* description / construction. An internal description has to consider all given concepts of a construction and to re-construct the build construction out from the inside. An *external* description is realized by an external observer of the construction knowing the rules of construction. A full polycontextural description has furthermore to take into account the complementarity of internal and external descriptions of its constructions.

It reads as follows:

> the operationality between operator and operand from the view of the operation,
> the operationality between operator and operation from the view of the operand,
> the operationality between operand and operation from the view of the operator.
> And:
> the operationality between operator and operand from the view of the position,
> the operationality between operator and operation from the view of the position,
> the operationality between operand and operation from the view of the position.

And so on.

All programming paradigms are conceived as a single, mostly highly complex and mixed unity, thus founded in uniqueness, represented as "1". This epistemological and semiotic closeness of the unique language is producing a special kind of double blindness: the blindness of its uniqueness and the blindness for its environment.

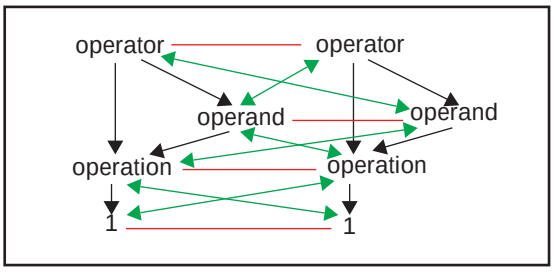
The blindness for its uniqueness is called the *Blind Spot* of the system. The system is not aware of its uniqueness. But even more, from a morphogrammatic point of view, the system is not aware of its location or of the fact of its locatedness. It is not realizing that it is occupying a place, i.e., a locus.

This monolithic uniqueness is called the mono-contexturality of the system. Blindness for itself is excluding the possibility for reflectionality, excluding self-observation, introspection and self-thematization.

The ability of an actional system to perceive and realize reflectionality is defining its *reflectional abstraction.*

2.3 Blindness for the Others: No interactionality

Blindness for the Others is excluding togetherness. In technical terms interactivity, co-operation and co-creation are not accessible in this world-view.



Interactivity, which is not changing the structure of architectonics, can be seen as a kind of reflectionality, reflection-onto-others. In other words, with a stable architectonics which is excluding metamorphosis and evolution / emanation, both concepts are complementary. That is, reflectionality can be seen as an interactivity in the modus of replication into itself. Both activities are complementary to each other and have to be distinguished properly. In polycontextural logic interactivity is mainly realized by different kinds of *transjunctions*. But interactivity is a general concept and is not reduced to logical operations only. In the terminology of super-operators of ConTeXtures interactivity is represented by the operator "*bifurcation*".

Interactional abstraction

Interactivity is not based on communication as information processing or message passing (Paul Dourish). It can be described as the action of addressing an addressee which is able to accept the addressing by its own addressable structure. After having been addressed and the addressing is accepted by the addressed and the addresser has recognized the acceptance of being addressed and the addressing is thus established, information can be exchanged between agents in the sense of processual communication (MAS, MIC).

Such a mutual action is realizing the *interactional abstraction*.

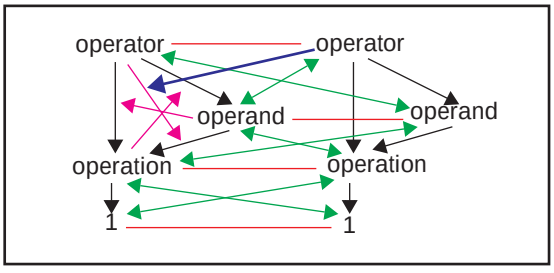
2.4 Thus, no interventionality

Without reflectionality and interactionality there is no *interventionality* and *interlocutionality*.

Interaction as reflection: reflectional interactivity (intervention)

Reflectional interactivity can be understood as an interaction unto the reflectional patterns of a neighbor agent or into the acting agent itself, therefore it can be called intervention and self-intervention.

Interventions are anticipating the behavior of an agent and try to influence it and to change its plans and motivations maybe to avoid conflicting situations.



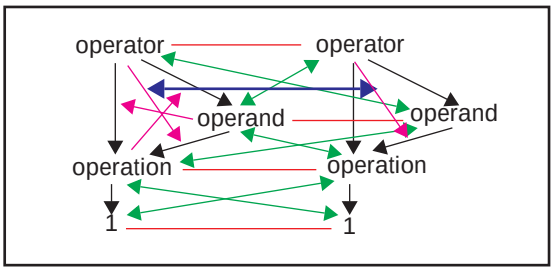
Intervention is re-programming the reflectional system of the neighbor system and not the system itself. The self-image of the neighbor system is re-programmed and not the system itself as it appears in an interactional context to the interacting agent and also not as the reflectional image of the neighbor in the internal environment of the agent.

Interventional abstraction

Because there is no direct access to the programming structure of another agent the perceived actions of the agent has to be analysed and the structure of the behaviorial pattern has to be discovered and abstracted from its communicational contents. With this reflectional knowledge about the behavioral patterns of the co-operating agent actions can take place with the intention to change the structure of this behavioral patterns. Thus, interventionality of an agent towards another agent is based on indirect reflectional interactions.

2.5 And no interlocutionality

Reflection as interaction: interactional reflectionality (interlocution, anticipation)



Interactional reflectional can been seen as a mutual interaction between two reflectional agents. Plans, motivations and strategies are directly involved with the aim to interact or change each others intentions and self-interpretations.

Interlocutional abstraction

The interlocutional abstraction is uncovering the dialogical and polylogical structure of a conversation between reflectional / interactional systems. This structure is the invariant pattern which is abstracted, discovered, identified and named, out of the communication process. Thus interlocutional patterns have to be filtered out of the stream of information to be recognized and applied for interlocutional interactions. Interlocution, thus, is a highly reflectional interaction of togetherness of anticipating agents.

3 General design of a contextural programming language

3.1 General tectonics of ConTeXtures

The tectonics of the *layout* of ConTeXtures can be introduced as the following heterarchic order of language levels.

1. General polycontextural **positional matrix**

Is studying the general *architectonics* of polycontexturality designed by its *complexity* (m) and *complication* (n). Excluding the dynamics of evolution and emanation of the design. There are *types* of matrices: under- (m<n), over-determinated (m>n) and balanced (m=n) matrixes. We are restricting ourselves in this study to *balanced matrices*.

The general matrix is represented as a structural *matrix* but also as a *diagram*, emphasizing the procedural aspects of the constructions.

2. General **templates** of the matrix

Is studying the general *templates* or scenarios of the distribution of systems over the matrix including its *interactivity* and *reflectionality* as interpretations of complexity and complication, here restricted to the balanced case. Other categories of the templates are *intervention* and *interlocution*.

3. General **patterns** of the templates

Is studying the structure of the general patterns of the templates of the matrix involving the distribution of the super-operators as specifications of the types of interactivity and reflectionality of the general design.

4. General **configurations** of the patterns

Is studying the general configurations the different programming *styles* (paradigms) and the multitude of *topics* (data structures, sorts, types).

Because the different programming paradigms are realized simultaneously and at once in different contextures, this dissemination of programming paradigms in ConTeXtures is called poly-paradigm. In contrast, multi-paradigms are known in mono-contextural programming languages as the possibility to deal with different programming styles in one language.

Additional, the general distribution and mediation of the specific *topics* (Boolean, Numeric, Symbolic, etc.) of the patterns has to be studied. Repetitions of the same topics in different contextures are called *mono-form* topics. Otherwise *poly-form* topics.

5. General **constellations** of the configurations

Is studying the combinations of the concrete realizations or *instances* of topics as operators and operands, functors and values, etc. in different contextures. E.g., Boolean: (and, or, trans), lists: (nil, cons, car) or poly-topics: (and, cons, one).

6. General **derivations** of the constellations

Is studying the intra- and trans-contextural derivations and computations of the complex contextural programming system.

Control structures may depend on styles and topics.

3.2 General epistemic activities of ConTeXtures

sketch-horizons:

The operator *samba* is sketching or designing the horizon of the computational constellation, that is, the general structure of the avouched textuality under consideration. As *lambda* is the operator for abstraction, samba, i.e, *samba's*, is the operator for thematization. Samba's is, as a term and an operator, at once, a singular unity and split into its multitudes. Designing the horizon is including the *complexity* and *complication* of the designed system. One of the relations between complexity and complication can be classified as *over-*, *under-* and *balanced*.

design-architectonics:

It is crucial to know how the fundament of the architectonic is conceived or build. Its graph-combinatorics can be a constellation from a linear to a star order, thus, *linear*, *tabular* and *circular*. Designing a horizon of programming is realizing the demand for reality construction by programming as a stronger approach than problem solving.

thematize-scenarios:

The super-operators are defining the interplay between different contextures of the designed scenario. The main modi of this interplay are *interactivity* and *reflectionality* and locally *iterativity* (computations). Other dimensions, like *intervention* and *interlocution* (anticipation), are possible, but are not explicitly included in this sketch.

A further concretization of the language is given by the programming *styles* chosen and the selected *topics*.

- *styles:* The different programming styles or paradigms, distributed over the mediated contextures, have to be addressed. The chosen styles can be in mono- or poly-style constellation. Styles are the paradigms of functional, imperative, object-oriented, contextural, logical and reflectional programming, etc.
- *topics:* The different operators, distributed over the mediated contextures, have to by thematized, taken into consideration. The operation *thematize* is defining the topics (mono- and poly-topics) to be brought into the scope of programming. Topis are the sorts of the language, like Boolean, Number, Symbolic, Class, Relation, etc.
- *contextures*: Contextures can be named reflectionally and interactionally, thus introducing some kind of self-reference into the contextural programming system.

identify-contextures:

Elementary contextures with and without interactive and reflectional connections to other intra- and trans-contextures are focussed. Identify-contextures is a function which is decomposing the complexity of the polycontextural situation given by the thematizing operation into its intra- and trans-contextural parts or modules. To each isolated or interacting contexture corresponds intra-contexturally a Lambda Calculus based ARS-system inheriting the reflectional and interactional distribution of the contextures.

local ARS systems:

define-operations: define <Name of Abstraction>

- Abstractions may be given a name explicitely, matching the general human understanding of `abstraction' as `to give something a name'.
- Abstractions may contain more than one lambda expression in the body to be evaluated.

abstract-functions: lambda <List of Parameters>

- Applications may contain more than two lambda abstractions including several arguments passed to the operator.

propose-statements: {Statements}

3.3 Bracket formulations of ConTeXtures

Full design of ConTeXtures

ConTeXtures

[*sketch* - ***horizon***$^{(m,n)}$
{ *complexity* | *complication* | { *over* - | *under* - | *balanced* } }
[*build* - ***architectonics***
{ *linear* | *tabular* | *circular* }
[*thematize* - ***scenarios***
{ *reflectional* | *interactional* | *interventional* | *interlocutional* }
[*choose* – ***styles***
[{ *functional* | *imperative* | *object* | *logical* | *contextural* }
[*select* – ***topics***
[{ *Boolean* | *numeric* | *symbolic* | *class* | *relational* | *reflectional* | *actional* }
[*identify* - ***contextures***
(*define* - ***operations***
(*abstract* - ***functions***
({ *propose* - ***statements***)))]]]]]]]

Short design of ConTeXtures

[***ConTeXtures***
[*sketch* – ***horizon***$^{(m,n)}$
[*build* – ***architectonics***
[*thematize* – ***scenarios***
[*identify* – ***contextures***
(*define* – ***operations***
(*abstract* – ***functions***
(*propose* – ***statements***)))]]]]]

Intra-contextural as-abstractions

[*thematize* ***intra - statements - as - statements***
[*identify* ***contexture***
(*define* < ***Name of Abstraction*** >
(*lambda* < ***List of Parameters*** >
{ < ***Statement***$_1^1$ > ***as*** < ***Statement***$_1^2$ >
< ***Statement***$_2^1$ > ***as*** < ***Statement***$_2^2$ >
. . . . . .
< ***Statement***$_n^1$ > ***as*** < ***Statement***$_n^2$ > }))]]

[*thematize* ***intra - name - as - name***
[*identify* ***contexture***
[*define* < ***(Name of Abstraction)***$_i$
as
(Name of Abstraction)$_j$ >
(*lambda* < ***List of Parameters*** >$_i$
{ < ***Statement***$_1$ >
< ***Statement***$_2$ >
. . . . . .
< ***Statement***$_n$ > })
(*lambda* < ***List of Parameters*** >$_j$
{ < ***Statement***$_1$ >
< ***Statement***$_2$ >
. . . . . .
< ***Statement***$_n$ > })]]]

4 What Contextural Programming is not?

"Only non interesting problems might be formulated unambiguously and solved completetly." Henry Poincaré

Mathematics vs. poly-mathematics

Strictly, contextural programming is not a part of *classic mathematics*, nevertheless it depends strongly on its established techniques and methodologies which are used/abused and transformed for polycontextural reasons. Thus, the project of contextural programming is in no sense against anything proved useful and established in classic mathematics and programming theories and languages. Contextural programming tries to surpass conceptual limits of the known approaches, simultaneously from the inside and the outside of classic paradigms of thinking, formalizing and programming.

Category theory

Also *category theory* is able to translate between the multitudes of different programming activities and language designs and to act as a unifying organizational model, poly-contextural programming is neither believing in nor opting for the ultimate universality of trichotomous-triadic concepts of category theory (Joseph Gogen) or Peircean semiotics and even less in established dichotomous-dyadic formalisms of binarism and digitalism. In a first attempt, contextural programming has to be connected with the project of n-category theory (John Baez, Tom Leinster) and poly-mathematics (V. I. Arnold).

Multiple-valued logics

Despite the fact that there are some historical connections between *multiple-valued logics* and polycontextural logics (place-value systems, Gotthard Gunther), contextural programming is not based or depending in any sense on it. Polycontextural logic is not a multiple-valued logic but a theory and formalism of dissemination (distribution and mediation) of all kinds of logics and logical systems. Also there is some similarity, it shouldn't be confused with the project of Combining and Fibering Logics (Gabbay, Pfalzgraf), modal logical approaches or context logics.

Complexity first

To put it in a simplicistic phrase, the main difference between classical and contextural programming is this: the first starts with structural *simplicity* and complexity is a late construct, the second starts with architectonic *complexity* and simplicity is a form of reduction. The philosophical difference is this: the first is considered with being, concepts, classes, terms representing things, objects, meanings, the second is opting for thinking, i.e., the mechanism of thinking of thinking, in contrast to thinking of something. Thus the first is semiotical and *ontological*, even onto-theological (Derrida, Goguen), the second is *kenomic* and *contextural*.

Metaphor of Verstand und Sinnlichkeit

Metaphorically, contextural programming and computation is mediating cognitive and aesthetic principles together, i.e., Verstand und Sinnlichkeit (Kant). Each contexture is placed in the world, is taking place, to realize this situatedness it has to be distinguished from others which can be done by an indexical sign, like the sign for its color. Thus, color is visualizing the positionality of a programming system. Thus, conceptual and aesthetic principle in their interplay are defining contextural programming.

Reality construction

"The language also serves as a framework within which we organize our ideas about processes." (Sussman) Contextural programming languages are strongly supporting the conceptual activity of organizing ideas. Mono-contextural languages may serve to organize the big design of a single programming design realized by a hierarchically organized team. The poly-paradigm approach of poly-contextural languages is demanding for complex co-operations and co-creations of horizontally organized teams of different epistemological locations. Their interests are beyond *problem solving* and are guiding future-oriented programming designs of *reality construction*.

Is there any chance to realize contextural programming?

Strictly, contextural programming can not be *realized* in the framework of an existing programming language. There is also no chance to *realize* it on existing hardware. So, is the idea of contextural programming simply a solitaire "brain-fuck"?

Things are not as harsh as it sounds if we move from the claim of strict *realization* to the more realistic job of *modeling* and *simulating* polycontextural features in accessible hard- and software systems. The main obstacles of a strict realization is given by the interactional and reflectional super-operators. They simply don't have any existence in classic systems. But there is no reason why they couldn't be simulated, say in the same sense as multi-tasking is simulated and not realized on a single-task chip.

General Limits of Multi-Processor-Systems for PCL

The main reason why it is not possible to *realize* polycontextural computing systems with multi-processor systems is embedded in the definition of the ALU of those systems. ALUs are containing in their logic junctions and negations, say as NAND or NOR operations. They are, obviously, not equipped with transjunctional operators. Transjunctions are for PCL systems the main logical operators of interactivity. On the other hand, it is not possible in polycontextural logics to define transjunctions with junctions and negations only. Otherwise there would be a chance to build a polycontextural computing systems out of a combination of distributed processors, organized as a special kind of a multi-processor system with distributed conjunctions and negations, poly-NANDs and poly-NORs.

This statement is in strict contrast to the genuine approach of Gunther in his main paper *"Cybernetic Ontology"* (1962) where transjunctions are defined by conjunctions and negations only. A reduction which seems to contradict his own aim to deliver a cybernetic theory of subjectivity. Because this result says, that subjectivity is definable in objective terms only. The new distinction Gunther introduced to define subjectivity, the possibility of rejection in contrast to acceptance, is reduced by this transformation to acceptance only. Thus, subjectivity is reducible to objectivity. This may be true for dead subjective systems but not for living systems.

These two facts, restriction of ALU and non-definability of transjunctions by junctions and negations only, are forcing the attempt to realize polycontextural computation on multi-processor systems from the attempt of *realization* to the attempt to *emulate/simulate*, i.e., to model such processes.

The other chance would be to design new processor types, not necessarily based on electronics, which would be able to realize directly transjunctional operations. It seems, that there are no "meta"-physical obstacles for that.

From Ruby to Rudy

First steps...to Diamondize Ruby

This text is looking for the missing space in which the antagonism of vertical and orthogonal/transversal programming could meet and be realized. At the same time it aims to give an idea about polycontextural and morphogrammatic approaches to programming. Some hints are taken from AOP, others from comics of Ruby's intros.

1 Conceptual modeling between IS and AS

Well, programming is about conceptual modeling. GP

After an enormous impressive tour through modern programming paradigms, especially AOP and *Generative Programming* (GP), a more conceptual chapter tells us some experiences about categorization, classification, class building and concepts. Also this chapter is quite short and it has not to be specially representative for the whole study by Krizysztof Czarnecki and Ulrich W. Eisenecker it is nevertheless confirming its traditional position. *"Concepts can be regarded as natural modeling elements because they represent a theory about knowledge organization in the human mind. The relationship between concepts and object-orientation is apparent: Concepts correspond to classes." GP, p 736*

Without surprise in neither of their approaches, the "classic view", the "probabilistic view" and the "exemplar view" to conceptual modeling, anything would appear which could escape the fundamental *is-has-abstraction* of cognitive modeling. Without saying, there is not the slightest shining of the magics of the as-abstraction to experience and to enjoy. In other words, the general concept or paradigm of computation and programming is not touched; it remains its natural taboo.

1.1 The classic view

"According to the classic view, any concept (or category) can be defined by listing a number of necessary and sufficient properties that an object must posses in order to be an instance of the concept." GP, p. 724

The scheme is: "define *name* statements".

1.2 The probabilistic view

"In the probabilistic view, each concept is described–just as in the classic view–by a list of properties, that is, we also have a single summary description of a concept. The main difference between them is that inn the probabilistic view each feature has a likelihood associated with it." GP, p.727

The scheme is: "define *name* statements modulo propabilistics".

1.3 The exemplar view

"In the exemplar view, a concept is defined by its exemplars rather than by an abstract summary." GP, p.729

The scheme is: "define *name* statements equal exemplars".

Also things are much more complex in concreto, in all cases, the statements are characterizing the concept named by its name by the general scheme of: X is Y.

1.4 Relation between classes

The use of the is- and has-relation are common place in concept modeling, OOP and other domains and are only shortly remembered and not properly discussed.

In this paper I will restrict my study to *abstractions* and their special relations (is, has, and as). A new kind of abstraction, the *morphic abstraction*, which is producing morphograms, is introduced, too. Standard works on Conceptual Reasoning and Modeling, like Sowa, Wille, etc. are not mentioned in this context.

- **IS-Relation**

"*This relation has to do with* inheritance*: A dog is an animal. A car is a vehicle. Because a dog is animal it has all the features of animal and of course on top of that its own."*

- **HAS-Relation**

"In the system using delegation *described above we have seen that an object must have an instance of its super-class in its attributes in order to be able to delegate an unknown message to a higher level."*

"*Of course an object may have instances of several super-classes making it easy to implement* multiple inheritance*."*

http://www.aplusplus.net/bookonl/node69.html

Problems with IS- and HAS-relations

All that sounds well established and not entangled with any problems which could lead to a questioning of the concepts involved. But as usual things are not so simple at a second glance. Two largely unsolved main problems are well known: *polysemy* and *multiple inheritance*. The known strategies to deal with polysemy are all leading to an *infinite regress*. The proposed solutions to multiple inheritance are leading, taken seriously, to paradoxes and *contradictions*. And all in all, it is not available to *machine-readable* solutions as proposed by the *Semantic Web* demands mainly because of the amount of human depending *negotiations*.

As a first step to enhance the classical concept of ontology which is behind the OOP approach I propose to introduce a new kind of relation/abstraction: the *AS-relation*.

1.5 AS-Relation

An object X, thematized as an object Y, is an object Z.

This wording correspondents to a general and neutral version of the as-abstraction. The statement *"A dog thematized as a dog is a dog"* is a realization of the form "X as X is X". It is easy to understand, that the classic ontological identity formula "X is X", which is used as the base of "everything", is an abbreviation of the reflectional form "X as X is X".

But *"A dog as a weapon is a danger."* is realizing the form: X as Y is Z.

This classic-ontological way of thinking, the is-abstraction, is guiding everything in computer science and programming and is not restricted to the case of OOP.

But in practice, the real life of programming looks quite different. It starts even at the very beginning of computing with the violation of its ontological identity principle. Take the use of programs AS data and the use of data AS programs. Or the interpreter as a program and the program as an interpreter. No problem! Yes, but also no theory of this practice. ISIS: X *is* X, ASIF: X *as if* Y, AZZA: X *as* Y, NINI: *Neither* X *nor* Y, more at:

http://www.thinkartlab.com/pkl/media/SUSHIS_LOGICS.pdf

An early application of AZZA to existential-therapy as Diamond Strategies:

http://www.thinkartlab.com/pkl/nlp-work/Deconstruction&DiamondStrategies.pdf

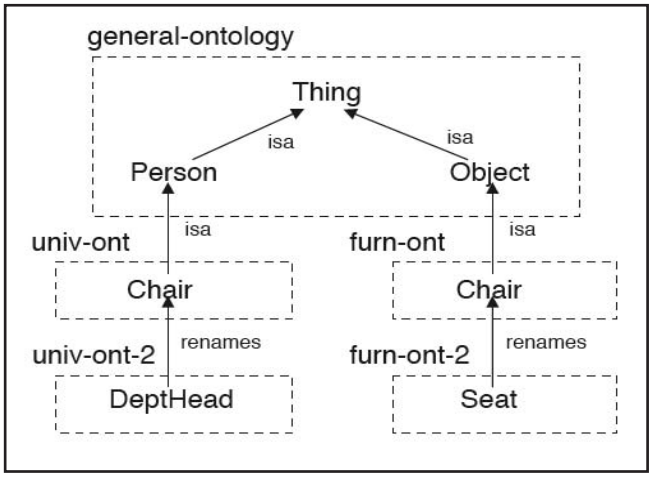
2 Polysemy: Conceptual modeling between abstraction types

One-step thinking leads to 1000-step disasters.

2.1 Polysemy in is-abstraction mode

The main principle of ontology is demanding for disambiguating the polysemy of the used terms. The simplest and historically oldest method to do this is given by *renaming* the terms. This is working perfectly in a very small world–where nobody lives.

> *The problems of synonymy and polysemy can be handled by the extension mechanism and use of axioms. An axiom of the form P1(x1; : : : ; xn) $ P2(x1; : : :; xn) can be used to state that two predicates are equivalent. With this idiom, ontologies can create aliases for terms, so that domain specific vocabularies can be used.*
>
> *For example, in Figure 3.1, the term DeptHead in OU2 means the same thing as Chair in OU due to an axiom in OU2. Although this solves the problem of synonymy of terms, the same terms can still be used with different meanings in different ontologies.*
>
> http://www.cs.umd.edu/projects/plus/SHOE/pubs/#aaai2000



There are many open questions left. How does it fit together to have an ontological relation “isa” and an obviously linguistic operation “rename”? To bring the modules furn-ont and furn-ont2 and also univ-ont and univ-ont2 together we need at least a mediation by a third module, which is reflecting the terminology of both. But this linguistic ontology would produce itself similar possibilities of polysemy.

Do it again: infinite regress of renaming

There is no reason to not to start the game of polysemy again with the term Seat as furniture and Seat as seat, e.g. position, in the hierarchy of a department. And we can disambiguate this polysemy again with the help of the term Chair. A seat as department is a chair and a seat as furniture is a chair. And now we can turn around as often as we want. Or we can enlarge the chain of renaming with Seat as Seat, the Portuguese car manufacturer SEAT or the Cafe Bar SeaT or Arthur's Seat in Edinburgh and so on…If something is working for my tiny household it shouldn’t be trusted for more.

Extension of ontologies by renaming is not violating the principle of verticality, that is hierarchy. Therefore, the tree is growing and with it its computational complexity. It becomes obvious that the procedure of renaming is part of the broader activity of *negotiation* which is not part of machine activities.

2.2 Polysemy in as-abstraction mode

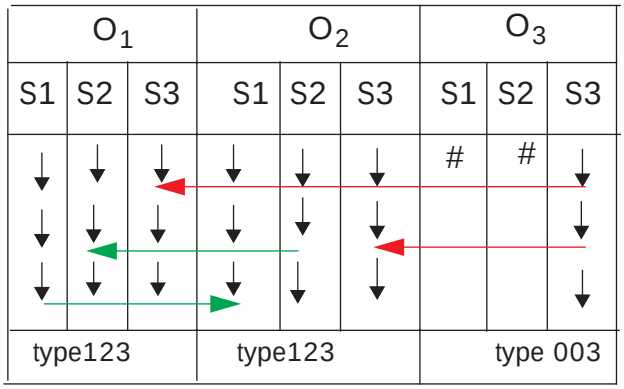
A reflectional analysis of polysemy using the as-abstraction is an analysis of the semiotic actions or behaviors of agents which is leading to the phenomenon of polysemy and its possible conflicts with other semiotic or logical principles. Therefore, such an analysis is more complex, because it has to describe the situation intrinsically, that is from the inside and not only externally from the outside of an external observer.

Mono-contextural introduction of "isa":
A: Chair is part of a furniture ontology,
B: Chair is part of a department ontology,
C: Chair is part of a vocabulary ontology.

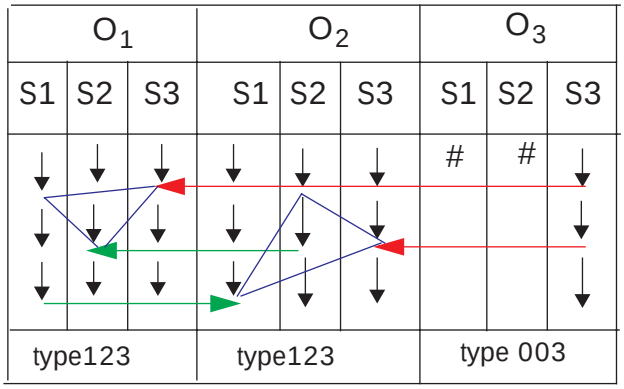
Poly-contexturally we have to translate these is-relations into following as-relations:
O1S1: Chair as such, that is, as an object "Chair",
O2S2: Chair as such, that is, as a person "Chair",
O3S3: Chair as such, that is, as a token "Chair".

Here, "as such" means, that the ontologies *Person*, *Object* and *Vocabulary* can be studied and developed for their own, independent of their interactivity and reflectionality to each other but mediated in the constellation of their poly-contexturality, that is, their distribution over 3 loci.

Interpretations of as-relations:
Voc O3S3 in Furn O1S3 : The token "Chair" as used to denote the object "Chair",
VocO3S3 in Dept O2S3 : The token "Chair" as used to denote the person "Chair",
Chair O2S2 in Dept O1S2 : The object Chair as used in the person ontology Dept,
Chair O1S1 in Furn O2S1 : The person Chair as used in the object ontology Furn.



Reflectional situations:
Chair O2S2 as in Dept O1S2:
System O1S1 has in its own domain space for a mirroring of O2S2. This space for placing the mirroring of O2S2 is the reflectional capacity realized by the architectonic differentiation of system O1. In other words, O1 is able to realize the distinction between its own data and the data received by an interacting agent. Data are therefore differentiated by their source, e.g. their functionality, and not only by their content.
Chair O1S1 as in Furn O2S1:
System O2S1 has in its own domain space for a mirroring of O1S1.



A (re)solution of the problem

The solution of the (new) problem is in the (old) problem which the (new) problem is the (old) solution.

1. The department Dept for itself has no conflict with polysemy. This conflict between Dept and Furn is mediated by the Voc. That is, the Person of the Dept as Chair is a person and nothing else.

2. The furniture Furn for itself has no conflict with polysemy. This conflict between Furn and Dept is mediated by the Voc. That is, the Chairs as objects of the Furn are chairs and nothing else.

3. The vocabulary Voc for itself has no conflict with polysemy between Dept and Furn.

4. The meaning of the polysemic situation is realized by

Meaning of (O3S3) = interaction of (O1S3, O2S3)

The conditions for a conflict arises exactly between:

O1(S1,2,3) and O2 (S1,2,3) mediated by O3S3 as visualized by the blue triangles.

Both Furn and Dept are using Voc and both are using the string Chair. Both are different and are mapping the Voc differently relative to their positions, thus the Voc has to be distributed over different places according to its use or functionality. The Voc used by Furn is in another functionality, i.e., as-relation, than the Voc used by Dept.

4. Until now we have not yet produced a contradiction but only a description of the situation of polysemy, that is, the necessary *conditions* for a possible ontological contradiction. A user-oriented or behavior-oriented approach to the modeling of polysemy has to ask "For whom is there a conflict?". Thus, polysemy is not conflictive per se but can occur for a user as a conflict.

5. Therefore, we have additionally to the semantic and syntactic modeling of the situation to introduce some pragmatic instances. In our example this could be the user of a *Query* which is answering the *User* in a contradictional manner.

Query´s contradiction

Thus, we have to deal with the contextures: (Query, Voc, Furn, Dept).

In the classic situation the Query answers with a logical conjunction of Chair as Person and Chair as a Department member, which are logically excluding each other and therefore producing for the User a contradictorily answer. Logic comes into play also

for the polycontextural modeling, but here conjunctions too, are distributed over different contextures. And therefore, there is no contradiction but separation. A contradiction occurs only if we map the complex situation all together into a single contexture. If we give up all the introduced ontological distinctions of polycontexturality and reducing therefore our ontologies to a single mono-contextural ontology we have saved our famous contradiction again. But now, this contradiction is a product of a well established mechanism of reduction and not a fallacy or a problem. And sometimes it isn't wrong to have it at our disposition.

Extension by mediation

The procedure of renaming can now be understood as an accretive ontology extension. To change from Chair as a furniture to Seat and from Chair as Dept to DeptHead is not only a linguistic procedure of renaming in the vocabulary it is also the use of two other ontologies in which these terms are common.

From the point of view of the new ontologies the conflict between Furn and Chair becomes obvious and transparent as a linguistic conflict of using a Voc. Only from the point of view of DeptHead and Seat the conflict appears as a conflict of synonymy. From the positions of Chair as Furn and Chair as Dept their is simply a conflict per se. Not opening up the possibility of an insight into its structure and kind of conflict and therefore there is also no chance for a solution of the conflict.

A solution, thus, as a re-solution, is possible only in the mode of as-abstractions realized in a polycontextural setting.

Comparison

The renaming procedure in the *is-abstraction mode* sounds very simple and intuitive compared to the proposed resolution of conflictive polysemy in the as-abstraction mode. The prize of the simple, one-step solution is, that it works only ad-hoc. As a conception it runs into infinite regress and at the end doesn't escape contradictions and confusions. These situations appear very quickly if applied to polysemic situation in the *Semantic Web* approach. First, the massive size of the Internet allows endless procedures of renaming, second, the necessity of *negotiations* in the process of renaming contradicts the aims of machine-readability of semantic procedures. Again, one-step solutions are producing 1000-step problems.

The *as-abstraction mode* is at a first glance quite difficult to understand and its exposition is still new and has yet to be improved. But there is some insight that this approach, also only exemplified and not yet formalized, is not only not a one-step ad-hoc convention but a functioning conceptual modeling of the complex situation of computational polysemy. That is, this approach is close to a feasible and finite and therefore, machine-readable design of complex interactions suitable as a mechanism to deal with the logics and semantics of the Semantic Web Vision.

http://www.thinkartlab.com/pkl/media/DERRIDA'S MACHINES.pdf

Question: Seth Russell: www-rdf-logic/2001Jul/0065
I think I've heard it said that the web must be monotonic. Have I misheard?
If not, then why must the web be monotonic?
Answer: Pat Hayes: www-rdf-logic/2001Jul/0067
Good question. The answer is controversial, but seems to me to be clear. First, its not the Web that is monotonic (whatever that would mean) but the reasoning from Web resources that must be monotonic. And the reason is that it - the reasoning - needs to always take place in a potentially open-ended situation: there is always the possibility that new information might arise from some other source, so one is never justified in assuming that one has 'all' the facts about some topic (unless you have been explicitly told that you have.) Nonmonotonic reasoning is therefore inherently unsafe on the Web. In fact, nonmonotonic reasoning is inherently unsafe anywhere, which is why all classical reasoning is monotonic; this isn't anything particuarly new.[...]
http://robustai.net/papers/Monotonic_Reasoning_on_the_Semantic_Web.html

3 Dimensionality of AS-abstractions

Abstraction is all there is to talk about: it is the object and the means of discussion.
Guy L. Steels

3.1 Sentence vs. textures

The name giving process is identifying its object and installing the laws of identity, thus these name givers are also called "identifiers". Like the lambda abstraction, which is defined or introduced intra-contexturally, the new samba abstraction is a trans-contextural operation. To distinguish it from the lambda abstraction, it should be called *samba thematization*.

Philosophically, to give a name is a special linguistic operation of the general mode of thematizing. Thematizing is more textual, to give a name is propositional or sentential. It is connected with the concept of a sentence as a statement. The philosophy of the lambda calculus is even stressing this further to the point, that the definition of a sentence is build by naming. To give a name is fundamental for the lambda calculus and radicalized, brought to the point by A++.

Hermeneutics and in radicalizing it, deconstructivism, tried to surpass this restriction and to focus more on texts, intertextuality, interpretability, iterability and ambiguity in contrast to well-formed single isolated sentences and propositions. In this sense poly-A++ can be considered as a further extension of the lambda calculus not from the inside but by distribution of the very idea and apparatus of the lambda calculus over different loci, empty places. Surely not in changing at all anything of the lambda calculus itself, but in disseminating it over the loci of the graphematic matrix.

Every programming language must somehow provide a `name giving' mechanism. Thus, every polycontextural programming language-system must somehow provide a general '*thematizing*' mechanism as a general feature allowing disseminated *'name giving' mechanisms* which each of them *allows to call procedures or functions and have the possibility to refer to variable*s inside the 'name giving' systems and between different 'name giving' systems.

A '*name giving*' procedure is also an *identifier*. To be able to identify something it has to be separated from its environment, but something can be separated from others only if it can be identified. We don't want to go into this paradoxical situation which is nevertheless the beginning of all formalism at all. But it should be mentioned that to identify something is including also a semiotic-ontological principle of identity: the named has not to be changed in the process of its naming. To name is to identify and not to change. But this is true only for the very special class of identical beings. It doesn't apply for living systems and even quantum physics is running into some troubles with this identity principle.

Abstraction as giving something a name is identifying something as something; it should be called *is-abstraction*. In contrast, the *as-abstraction* is identifying something as something else. That is, as-abstraction is thematizing something as something in a specific context (situation, constellation, environment). As-abstraction, i.e., thematization, is naming something as something and giving the context of its identification. The context of identification is designed by the architectonics of polycontexturality. Abstraction as giving something a name is emphasizing the act of classification in contrast to creation. As-abstraction is not naming something pre-given but evoking new creations.

as-abstraction = [evocation, contextualization]
is-abstraction = [identification, classification]

3.2 As-abstraction and object-schemes

A first analysis shows the 2-dimensional structure of the as-abstraction. The as-abstraction is involved into the process of thematization. One dimension is the semantic-ontological dimension, which is inherited by the is-has-component of the as-abstraction. The second dimension is produced by the contextualization of the semantic distinctions, something-as-something-else-is-something-else.

An object is always, in a reductive parlance, a n-tuple of determinations.

The formula: X as Y is Z, surely is only an abbreviation for the general scheme:

$$\left[object^{(n)}\ as \begin{bmatrix} 1-object \\ 2-object \\ ..... \\ n-object \end{bmatrix} is^{(n)} \begin{bmatrix} 1-object-1 \\ 2-object-2 \\ ..... \\ n-object-n \end{bmatrix} \right]$$

Thus, the separation of OOP and AOP in respect to Concerns can be dynamized to a more flexible approach with the introduction of the concept of *dynamic object-schemes.* There are some similarities to the Metapattern approach, too.

If we start OOP with dynamic and complex *object-schemes*, queer or horizontal consideration can be involved without problems. An object is always based on an object-scheme, for OOP it is simply a 1-object-scheme, thus for short, an object. Thus, an object is an object, and nothing else.

$$\left[a\ dog\ \text{as} \begin{bmatrix} a\ dog \\ an\ animal \\ a\ wapon \\ ..... \\ robot \end{bmatrix} \text{is}^{(n)} \begin{bmatrix} a\ dog \\ a\ creature \\ a\ danger \\ ..... \\ a\ toy \end{bmatrix} \right]$$

Obviously, all thematizations of "dog" are producing different characterizations of the is- and has-structure (of the concept "dog"), and have different consequences for philosophers, animal rights people, police and army, toy industry, and so on. For a programmer it is only a question of the dimensionality of the objects, or abjects.

Thus, the concept or object "dog" is overdetermined, object$^{(n)}$, depending on changing use, contexts, thematizations, etc. In other words, a complex object is not pre-given but constructed in societal interactions and reflections. As an ontological consequence, the statement "a dog is a dog", emphasizing on the reflexivity of the class relation, is on 1-object-scheme level, but understood in the sense of *"a dog as such"* is on the level of 0-object-scheme belonging to a Platonic World. Multi-dimensionality of the as-relation should not be confused with the n-adic relations of the is-abstraction as it is well established.

The above introduction of object-schemes is only a first step which is limited by the analysis of *ordinary language*. Ordinary language as such is not offering enough formal distinctions to develop the whole range of the dynamics of as-abstractions. Complexity and dynamics of ordinary language is mainly realized by its semantics. To realize object-schemes more properly we need not only syntactics but also the architectonics and tectonics of language. Despite interesting work by linguists and philosophers about *metaphors* and *polysemy*, analysis is best realized with formal languages. Scriptural systems of polycontexturality with their heterarchic tabular architectonics and disseminated tectonics are able to incorporate complex chiasms between its levels.

4 Contextu(r)alizing the AS-abstraction

Abstraction as giving something a name is emphasizing the act of classification in contrast to the act of co-creation. ThinkArt Lab

Following the *architectonics* and *tectonics* of the lambda calculus and ConTeXtures some of the combinatory possible as-constructions are presented below.

Again, the as-abstraction can be defined *intra-contextural*, contextualizing names from context to context inside a contexture or *trans-contextural*, contexturalizing names between different contextures. The latter application of the as-abstraction is crossing and bridging different contextures in a single or multiple way.

A first step forward in formalizing polycontextural programming is: From "define name" to "define a $name_i$ as a $name_j$ between different contextures".

This AS-function I have introduced as an extension of the OOP strategies. But it is introduced at the very beginning of the lambda construction "define name" as it appears on the base of the proemial relationship. Not only contextures can be named and names be contexturalized but also names can be named as other names between and inside of contextures in the sense that $name_j$ refers to another context or contexture than $name_i$, both belonging to the complexion involved in the play. Therefore, the referentiality and transparency of ConTeXtures is not restricted to any hierarchy of tectonics. Levels and meta-levels of reflection are connected by means of proemiality realizing its structural rules of exchange, order and categorial correctness (coincidence) avoiding wild jumps, structurally possible, but not analyzed in this context.

4.1 Modus I: A *name is a name* in a single contexture

$$\left[\begin{array}{l} thematize\ mono-contextural\ \textbf{is} \\ \left[\begin{array}{l} identify\ contexture \\ \left[\begin{array}{l} define\ (Name\ of\ Abstraction)_j \\ \left(\begin{array}{l} lambda < List\ of\ Parameters > \\ \left\{\begin{array}{l} < Statement_1 > \\ < Statement_2 > \\ \quad \ldots\ldots \\ < Statement_n > \end{array}\right\} \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right]$$

Abstraction as "giving a name" is defined intra-contextualy as identifying its named object. This abstraction is on the base of the lambda calculus and programming languages based on it, like LISP, SCHEME, ARS.

Because of its monocontexturality the two headers "thematize" and "identify" can be omitted in the classic use.

But it remains nevertheless a precondition of the whole construction. Everything else is boxed into this little construction of (head + body).

A short remainder of the syntax in EBNF-Notation for ARS

```
<contexture> :: = <expression> | | <position>=1

<expression>  ::= <abstraction> |  <reference>  | <synthesis>

<abstraction>  ::= '(' define <variable> <expression> ')' |
                                  '( lambda (' {<variable>}')'  |
                                 <expression> { <expressions> }')'

<reference>   ::= <variable>
<synthesis>    :: = '(' <expression>  { <expression> }')'
<variable>    ::= <symbol>
```

EBNF-syntax plus "| |": transcontextural parallelity

4.2 Modus II: *Statements as statements* in a contexture

Contextual use of the as-abstraction is possible, even in the classic lambda calculus, between statements: $statement_i$ as $statement_j$ in a fixed contexture of naming. That is in the domain defined by the calculus.

$$\left[\begin{array}{l} thematize\ intra-contextural\ \mathbf{as} \\ \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define < Name\ of\ Abstraction > \\ \left(\begin{array}{l} lambda < List\ of\ Parameters > \\ \left\{\begin{array}{l} < Statement_1^1 > as < Statement_1^2 > \\ < Statement_2^1 > as < Statement_2^2 > \\ \quad \cdots\cdots \\ < Statement_n^1 > as < Statement_n^2 > \end{array}\right\} \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right]$$

Thus, this use is restricting dynamics to intra-contextural changes of contexts, but excluding crossings to other contextures. Connections of this kind of abstraction to context based lambda calculus, typed systems and fibred lambda calculi could be a natural application. The following syntax in EBNF notations should be read as first step approaches.

A short syntax of "statement as statement" in EBNF-Notation

<contexture> :: = <expression> | | <position>=1

<expression> ::= <abstraction> | <reference$_i$ as reference$_j$> | <synthesis>

<abstraction> ::= '(' define <variable> <expression$_i$ as expression$_j$> ')' |
'(lambda (' {<variable>}')' |
<expression> { <expressions> }')'

<reference> ::= <variable$_i$ as variable$_j$>
<synthesis> :: = '(' <expression> { <expression> }')'
<variable> ::= <symbol$_i$ as symbol$_j$>

ConTeXtures

$$\left[\begin{array}{l} sketch-horizon^{(m)} \\ \left[\begin{array}{l} build-architectonics \\ \left[\begin{array}{l} thematize-scenarios \\ \left[\begin{array}{l} choose-styles \\ \left[\begin{array}{l} select-topics \\ \left[\begin{array}{l} identify-contextures \\ \left(\begin{array}{l} define-operations \\ \left(\begin{array}{l} abstract-function \\ \left\{propose-statements\right\} \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

In ConTeXtures there are, at least, two abstractors in the play: the (classic) is-abstraction *select* and the polycontextural as-abstraction *elect*.The selector is defining and founding the "*define*"-operation, the elector is introducing the *identify*-operation which is electing a contexture. On a higher level, the operator "*thematize*" is defining the scenarios and score of programming. Chose style is introducing different programming styles. Select topics show which sorts and types are under consideration. At the end, we have the disseminated lambda calculi, "based" on the complexity and general structure of dissemination, the heterarchic *architectonics*. Thus, programming languages based on contextures are distributed heterarchically along the architectonics of ConTeXtures which can be seen as a distribution of ARS.

4.3 Modus III: A *name as a name* in a contexture

$$\left[\left[\begin{array}{l} thematize\ intra-contextural\ \textbf{\textit{name - as - name}} \\ \left[\begin{array}{l} identify\ contexture \\ \left[\begin{array}{l} define < (Name\ of\ Abstraction)_i\ AS\ (Name\ of\ Abstraction)_j > \\ \left(\begin{array}{l} lambda < List\ of\ Parameters > \\ \left\{\begin{array}{l} < Statement_1 > \\ < Statement_2 > \\ \ldots\ldots \\ < Statement_n > \end{array}\right\} \end{array}\right) \left(\begin{array}{l} lambda < List\ of\ Parameters > \\ \left\{\begin{array}{l} < Statement_1 > \\ < Statement_2 > \\ \ldots\ldots \\ < Statement_n > \end{array}\right\} \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right]\right]$$

The steps of interpretations of names, *name$_i$ as name$_j$*, can be made explicit by introducing the as-abstraction in the "name giving" procedure. Thus, the innocent identifier "define name" has to be enlarged to "define name$_i$ as name$_j$" inside a contexture.

A short syntax of intra-contextural *"name as name"* in EBNF-Notation

<contexture> :: = <expression> | | <position>=1

<expression> ::= <abstraction$_i$ as abstraction$_j$> | <reference$_i$ as reference$_j$> | <synthesis>

<abstraction> ::= '(' define <variable$_i$ as variable$_j$> <expression$_i$ as expression$_j$> ')' |
'(lambda (' {<variable$_i$ as variable$_j$>}')' |
<expression> { <expressions> }')'

<reference> ::= <variable$_i$ as variable$_j$>
<synthesis> :: = '(' <expression> { <expression> }')'
<variable> ::= <symbol$_i$ as symbol$_j$>

Also this scenario stays inside a single contexture it opens up a great flexibility of interchanging definitions of concepts. Applied to types it rules the changes between types, say from Boole to List, etc. inside a contexture. This change allows to apply several definition of an object at once. A "neutral" object, thematized in one context, can be dealt in another context as of type LIST and simultaneously as of type NUM in a further context. Thus, this kind of abstraction is introducing at the very beginning of its definition the possibility of dealing with contexts. The whole scenario can be iterated for several contextures opening up contexturally different kind of (intra-contextural) contextual systems. Not to confuse a multitude of contexts with systems containing a plurality of contextures. One is part of context logics, the other of polycontectural logics.

This kind of as-abstraction inside the naming procedure, allows to call-by-name the name itself in the modus of sameness. This self-call is still enclosed in a common contexture but deliberates the naming procedure to a context related self-naming. In the following, self-naming is deliberated step by step to the possibility not only to name a contexture but to the possibility of thematizing contextures by contextures. Naming, to give something a name, to make an abstraction within propositions is considered as a very special mode of thematization in textual constellations. Thus, thematization is introducing different strength of architectonic and tectonic self-referentiality.

4.4 Modus IV: A *name as a name* between contextures

$$\left[\begin{array}{l} thematize\ trans-contextural\ \textbf{name - as - contexture} \\ \left[\begin{array}{l} identify\ contexture_i \\ \left[\left[\left[\begin{array}{l} define\ (Name\ of\ Abstraction)_i \\ \left(lambda < List\ of\ Parameters >\right) \\ \left\{\begin{array}{l} < Statement_1 > \\ < Statement_2 > \\ \ldots\ldots \\ < Statement_n > \end{array}\right\} \\ elect\ name_j \end{array}\right]\right]\left[\left[\begin{array}{l} identify\ contexture_j \\ \left[\begin{array}{l} define\ (Name\ of\ Abstraction)_j \\ \left(lambda < List\ of\ Parameters >\right) \\ \left\{\begin{array}{l} < Statement_1 > \\ < Statement_2 > \\ \ldots\ldots \\ < Statement_n > \end{array}\right\} \end{array}\right] \end{array}\right]\right]\right] \end{array}\right] \end{array}\right]$$

A short syntax of trans-contextural "name as name" in EBNF-Notation

<contextures>$^{(m)}$ :: = <expression> || <position$_1$> || <position$_2$> ||...|| <position$_n$>

<expression$_i$ as expression$_j$>$^{(m)}$ ::= <abstraction$_i$ as abstraction$_j$> |
<reference$_i$ as reference$_j$> |
<synthesis$_i$ as synthesis$_j$>

<abstraction$_1$ as abstraction$_2$>$^{(m)}$ ::= '(' define <variable$_i$ as variable$_j$>
<expression$_i$ as expression$_j$> ')' |
'(lambda as lambda$_j$ (' {<variable$_i$ as variable$_j$>}')' |
<expression$_i$ as expression$_j$> { <expression$_i$ as expression$_j$> }')'

<reference>$^{(m)}$ ::= <variable$_i$ as variable$_j$> || <variable$_i$ as expression$_j$>
<synthesis>$^{(m)}$ :: = '(' <expression$_i$ as expression$_j$> { <expression$_i$ as expression$_j$> }')'
<variable> $^{(m)}$ ::= <symbol$_i$ as symbol$_j$> || <symbol$_i$ as expression$_j$>

This construction of the as-abstraction, i.e., thematization, is enabled to call by name from the viewpoint of one contexture the name in another contexture. It is not yet able to call another contexture as such but has access to the name in use of it. The name of the different contexture is defined by another abstraction than is the name of the calling name. To call a name of a different contexture is not a simple jump to another contexture because both the first and the second distinctions remain in use at once.

In the examples, I am not considering the structure of the mechanism responsible for the exchanges. This would introduce the well known *proemial relation,* short *chiasm*.

http://www.thinkartlab.com/pkl/lola/poly-Lambda_Calculus.pdf

A kind of a formula for chiasm (M, σ)

$$M^3 \triangleq M^1 \longrightarrow \sigma^1 \updownarrow M^2 \longrightarrow \sigma^2 \triangleq \sigma^3$$

or

$$\begin{array}{l} M^1 \longrightarrow \sigma^1 \updownarrow M^2 \longrightarrow \sigma^2 \\ \triangleq \qquad\qquad\qquad\quad \triangleq \\ M^3 \longrightarrow\qquad\qquad\quad \sigma^3 \end{array}$$

$$\left[\begin{array}{l} thematize\ chiasm\left(M, \sigma\right)^{(3)} \\ \left[\begin{array}{l} contextures^{(3)} \\ \left[\left[\left[\begin{array}{l} identify\ LC^1 \\ \left[\begin{array}{l} M \\ \left[\begin{array}{l} \sigma \\ \left[elect\ M^2\right] \end{array}\right] \end{array}\right] \end{array}\right]\right]\left[\left[\begin{array}{l} identify\ LC^2 \\ \left[\begin{array}{l} M \\ \left[\begin{array}{l} \sigma \\ \left[elect\ \sigma^1\right] \end{array}\right] \end{array}\right] \end{array}\right]\right]\left[\left[\begin{array}{l} identify\ LC^3 \\ \left[\begin{array}{l} M \\ \left[\left[\sigma\right]\right] \end{array}\right] \end{array}\right]\right] \end{array}\right] \end{array}\right]$$

4.5 Modus V: A *name as a contexture* in a polycontexturality

One of the most intriguing possibilities is the chiasm, interlocking mechanism, between a name of a contexture and a contextures of a compound contexturality. It wouldn't be much reasonable to treat this possibility in an is-abstraction formalism. Because the part (name) would be the whole (contexture) and vice versa. But a name as a contexture and a contexture as a name is not only a reasonable wording but a base for a working formalism.

$$\left[\begin{array}{l} thematize\ chiasm\left(\boldsymbol{name,\ contexture}\right)^{(3)} \\ \left[\begin{array}{l} contextures^{(3)} \\ \left[\begin{array}{l} identify\ contexture^{1} \\ \left(\begin{array}{l} define\ name \\ \left\{parameters\right\} \end{array}\right) \\ elect\ contexture^{2} \end{array}\right] \left[\begin{array}{l} identify\ contexture^{2} \\ \left(\begin{array}{l} define\ name \\ \left\{parameters\right\} \end{array}\right) \\ elect\ contexture^{1} \end{array}\right] \left[\begin{array}{l} identify\ contexture^{3} \\ \left[\begin{array}{l} define\ name \\ \left\{parameters\right\} \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

A short syntax of trans-contextural "name as contexture" in EBNF-Notation

<contextures>$^{(m)}$ :: = <expression$_1$> || <expression$_2$> ||...|| <expression$_n$>

<expression$_i$ as expression$_j$>$^{(m)}$ ::= <abstraction$_i$ as abstraction$_j$> |
<reference$_i$ as reference$_j$> |
<synthesis$_i$ as synthesis$_j$>

<abstraction$_1$ as abstraction$_2$>$^{(m)}$ ::= '(' <define$_i$ as define$_j$> <variable$_i$ as variable$_j$>
<expression$_i$ as expression$_j$> ')' |
'(lambda$_i$ as lambda$_j$ (' {<variable$_i$ as variable$_j$>}')' |
<expression$_i$ as expression$_j$> { <expression$_i$ as expression$_j$> }')'

<reference>$^{(m)}$ ::= <variable$_i$ as variable$_j$> || <variable$_i$ as contexture$_j$> ||
<contexture$_i$ as variable$_j$>
<synthesis>$^{(m)}$ :: = '(' <expression$_i$ as expression$_j$> { <expression$_i$ as expression$_j$> }')'
<variable>$^{(m)}$ ::= <symbol$_i$ as symbol$_j$> || <symbol$_i$ as contexture$_j$> ||
<contexture$_i$ as symbol$_j$>

Additional to the abstraction *"name as name between contextures"*, this abstraction between *name and contexture* delivers mechanisms for "calling" a contexture by name, thus enabling strict reflectional programming. A reflection of a system onto itself is possible in the modus of sameness as contrasted to the modus of equality (identity). In the case of equality not much more than paradoxes and endless iterations into hierarchies to avoid conflictive constellations are possible. In the modus of sameness, reflections of all kind are possible without producing any contradictions. But the price we have to pay is to move from equality to sameness and to loose the paradise of homogeneous uniqueness of the programming paradigm. In other words, we have to risk a jump between different contextures, not fearing the abyss of *dis-contexturality* between a multitude of different "roots".

Towards EBNF of EBNF: http://www.csci.csusb.edu/dick/papers/rjb99g.xbnf.html

Polycontexturality understood as fibered logics, made save with index categories at:
http://racefyn.insde.es/Publicaciones/racsam/art%C3%ADculos/racsam%2098_1/2004-pfalzgraf.pdf

4.6 Example of a transcontextural as-abstraction

An example of a transcontextural as-abstraction formulated in the framework of ConTeXtures is given below. The point in this example is not so much the shifts from one topic to another, say from "zero" to "nil" and to "flase", this could still be intra-contextural, but the fact that these different topics are located in different contextures, all having their own topics, too. Also this example shows a polycontextural constellation, its polycontexturality is limited to reflectionality, excluding interactionality. Thus, "nil" in contexture$_{1.1}$, not thematized in this example, is contexturally different from "nil" in contexture$_{1.2}$.

$$
\begin{array}{l}
samba^{(3)}(repl, \varnothing, \varnothing) - horizon \\
\left[\begin{array}{l}
poly - topics\ (Num,\ List,\ Boole) \\
\left[\begin{array}{l}
thematize\ (zero,\ nil,\ false) \\
\left[\begin{array}{lll}
\left[\begin{array}{l} identify\ contexture_{1.1} \\ \left(\begin{array}{l} define\ zero \\ \left(\begin{array}{l} lambda\ (\ f\ ) \\ \left(\begin{array}{l} lambda\ (\ x\ ) \\ x \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right] & \varnothing & \varnothing \\
\left[\begin{array}{l} identify\ contexture_{1.2} \\ \left(\begin{array}{l} define\ zero_{1.1}\ as\ nil \\ \left(\begin{array}{l} lambda\ (\ l\ ) \\ true \end{array}\right)\end{array}\right)\end{array}\right] & \varnothing & \varnothing \\
\left[\begin{array}{l} identify\ contexture_{1.3} \\ \left(\begin{array}{l} define\ nil_{1.2}\ as\ false \\ \left(\begin{array}{l} lambda\ (a\ b\ ) \\ b \end{array}\right)\end{array}\right)\end{array}\right] & \varnothing & \varnothing
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}
$$

Thus, "define *name*" is an abbreviation of "define *name$_i$* as *name$_j$*" with i=j.

– replication *repl,* in this example, is a metamorphic replication and not replicating isolated configurations.

Exchange relations:

– "define *zero*" is "define *zero* as *zero*", as the start of the levels. It could itself be produced by a predecessor level.

as: define *zero* in contexture$_{1.1}$ as *zero* in contexture$_{1.1}$

– "define *nil*" is "define *zero* as *nil*",

as: define *zero* from contexture$_{1.1}$ as *nil* in contexture$_{1.2}$

– "define *false*" is "define *nil* as *false*".

as: define *nil* from contexture$_{1.2}$ as *false* in contexture$_{1.3}$

This change of identity of the topics from one contexture to another by reflection/replication is producing a chiastic chain guaranteeing the connectedness of the step-wise reflection of the whole. This chain is ruled by reflectionality, produced by the operator *replication*. An interacting neighbor system, S1 or S2, could reflect in its contexture this fact of chiastic connectedness. It is of importance, that this chain is ruled by chiasm or the proemial relation with its components of order, exchange and coincidence relation. Otherwise the chain simply would not be a mediated chain but a summation or iteration of reflectional steps.

Other examples would model the constellation in an interactional way, involving different interacting contextures, or in a mixed way of interactional and reflectional realizations. All mediated together and enabled by the proemial relation. And the whole machinery of different types of as-abstractions can be involved together.

What is working for elementary situations like topics (Num, List, Boole) in the above example, can be applied to handle more concrete programming features like objects and aspects, primary and secondary concerns, security, logging, and others.

Some ideas of mixed modeling of iterability of Gödel constructions is developed at: http://www.thinkartlab.com/pkl/lola/Godel_Games-short.pdf

5 Object-Schemes as Morphograms

Abstraktion schwächt, Reflexion stärkt. Novalis

To be more consistent to my usual terminology I have to introduce *morphograms*.

In a strict sense, object-schemes are morphograms. Simply because object-schemes are the schemes of possible complex objects, but object-schemes as schemes are not themselves objects at all. Object-schemes are giving space to inscribe the complexity of objects independent of the content of it. That is, object-schemes are neutral to data-types, to sorts and other typing distinctions. Thus, neutral to objects.

But object-schemes, based on complex as-abstractions, are not yet giving information about the *structuration* of the complexity of the complex object. The list of different as-components in the determination of the complex object may look like a homogeneous space of as-attributes. This is not only quite un-realistic but also contradicts the plurality of contextures of a complex object.

Morphograms are accessible by the *morphic abstraction,* which in fact is a *subversion,* and are studied in the new discipline *morphogrammatics*. The morphic abstraction is abstracting the holistic structure of complex object-schemes. Thus, the whole as-abstraction is based on morphograms. Morphogrammatics are opening up the possibility of a "paradigm-free" scriptural system for conceptualization and programming.

(In linguistics and grammatology morphograms, esp. for Chinese and Japanese writing, are also called *logograms* or *ideograms*.)

Cf. http://www.uni-ulm.de/uni/intgruppen/memosys/desn22.htm

> *"But since we pointed out that every ontological datum of the world must be considered an intersection of an infinite number of contextures, the fact that any two data we choose to describe in their common two valued relations belong to one contexture does not exclude that the very same data may also apart from the contexturality chosen for our description belong separately to additional and different contexturalities."*
> Gotthard Gunther, Life as Polycontexturality.

What does it mean?

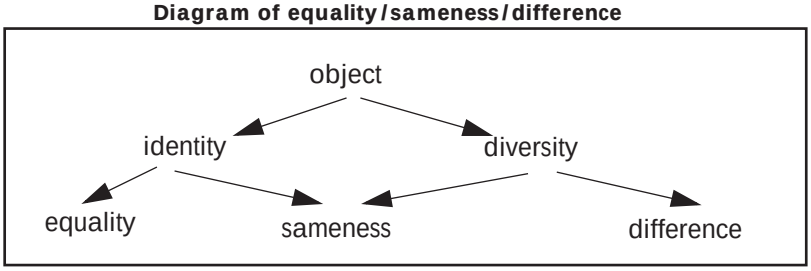
To speak and program about as-this-and-as-that treats the parts of the complexions, this-and-that, as neutral and homogene. There are no semantic-ontological differences between the parts involved. But this is not only an advantage in the sense of a broad homogeneity but also can turn into a disadvantage as a loss of structuration.

Thus, structured object-schemes are presented, inscribed by morphograms.

Concept modeling in a morphogrammatic sense is not rooted in an ultimate root as the unique origin of conceptualization, but in a multitude of beginnings and ends, mediated together by their morphogrammatics. Thus, there is no ultimate "Class Object" as the root of the system, but a dynamic interlocking mechanism of conceptual beginnings and ends based on pragmatic decisions. By the complexity of an object I understand the number of contextures involved. *Objectionality* thus is structured by the sameness (Gleichheit) and difference (Verschiedenheit) of its contextures. This distinction is quite intriguing because it is not based on equality and its logics. The German language allows to introduce differences into the distinction identity/diversity as Selbigkeit/Gleichheit/Verschiedenheit. Maybe *equality/equivalence/difference* and its iteration/accretion could help to introduce more *liberty* into the game of our thinking, freeing us from the necessity of killing what is different and not part of the scheme.

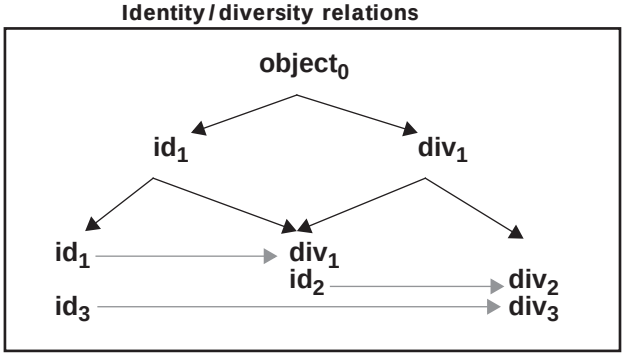
5.1 Diagrams of equality, sameness and difference

Diagram of equality/sameness/difference



The diagram show a general scheme of an extension of the difference of identity/diversity to the differences of equality/sameness/difference.

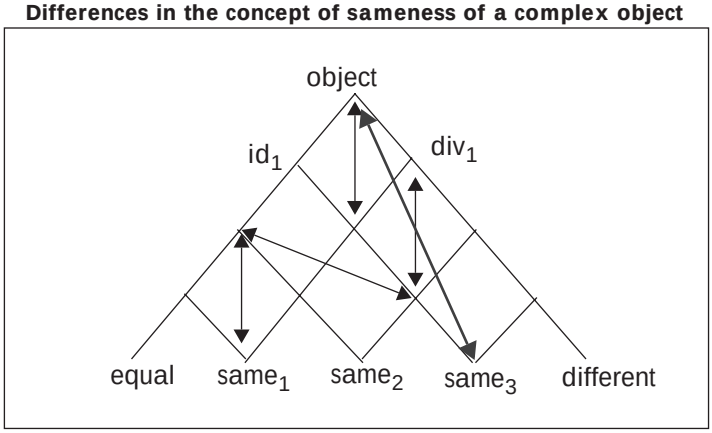
Identity/diversity relations



This extension can be modeled as a distribution of the original identity/diversity relation over 3 loci. Thus, iterating the difference: equality = {id_1, id_3}, sameness = {div_1, id_2}, difference = {div_2, div_3}. Obviously, all these id/div-relations are semantically founding a base of a logical system, delivering different negations.

Negations in polycontextural logics have two function: 1) *inversion* of the values (id/div) and 2) *permutations* of the subsystems involved. Thus, $N_1(id_{1.3}, div_1/id_2, div_{2.3}) = (div_1/id_2, id_{3.1}, div_{3.2})$. That is, the values of subsystem$_1$ are inverted to (id_1, div_1) => (div_1, id_1) and the subsystems$_{2.3}$ are permuted to subsystems $_{3.2}$.

Differences in the concept of sameness of a complex object



Different paths through the graph of the determination of an objects' complex identifying structure can be studied and linked to multi-negational operations.

Without doubt, the ambiguity can also be distributed over the terms "equal" or "different". Thus, different interpretations of the id/div-relation are possible. E.g, ($equal_1$, $equal_2$, same, $different_1$, $different_2$) or ($equal_1$, $same_1$, $same_2$, $different_1$, $different_2$). With additional terms for id/div-clusters new wordings are avaiable.

http://www.thinkartlab.com/pkl/media/SKIZZE-0.9.5-medium.pdf

5.2 An example: 3-fold objectionality

For a complexity of m=3, representing 3 viewpoints or aspects there are only 5 patterns of complexity of the object-scheme, that is, only 5 morphograms, possible. For m=4, 15 morphograms are introduced. The number corresponds to the Stirling Numbers of the Second Kind (1, 2, 5, 15, 52, 204, ...). Thus, for 6 viewpoints we get 204 patterns.

$$morph\left(object-scheme^{(3)}\right) = \begin{bmatrix} \triangle & \triangle & \triangle & \triangle & \triangle \\ \triangle & \triangle & \square & \square & \square \\ \triangle & \square & \triangle & \square & \bigcirc \end{bmatrix}$$

The 5 morphograms mg_i are inscribing the range of distribution of sameness and difference of viewpoints over 3 loci.

$$morph\left(object^{(3)}\right) = \left[mg_1,\ mg_2,\ mg_3,\ mg_4,\ mg_5\right]$$

$$refl\ \left(mg_i\right) \equiv mg_i,\ i = 1,3,5$$

$$refl\ \left(mg_2\right) \equiv mg_4,\ \ refl\ \left(mg_4\right) \equiv mg_2$$

$$Thus,\ refl\left(refl\ \left(mg_2\right)\right) = mg_2,$$

$$and,\ refl\left(refl\ \left(mg_4\right)\right) = mg_4$$

Reflector on Morphograms

A simple operator on morphograms can be introduced, the reflector *refl.* This reflector simply produces an inversion of the pattern, that is of the morphogram.

Some morphogrammatic equivalences, using letters:

Thus, refl[aab] = [baa] = [abb]

While, refl[abc] = [cba] = [abc]

The operator *refl* can be used to produce changes in the structure of object-schemes, say as a change of interests, priorities, etc., while not changing its complexity.

Object-schemes are implementing different general viewpoints, thus, the operator *refl* is a simple and elementary operator of changing the order and priority of viewpoints in a reflectional system. This change is not eliminating, reducing or augmenting viewpoints but only changing their priority and relevance. By convention we could apostrophe the first locus of a morphogram as the first viewpoint. Thus a repetition of the first in [aab] can be changed by the reflector *refl* into a repetition of the second and reducing the first's repetition to a single occurrence, as in refl[aab] = [abb].

Other operators exists, e.g., *reductors*, which are reducing the structure of morphograms to simpler morphograms, say, red[abc] = [aab]. Morphic abstractions are in some kind abstractions of abstractions. Abstractions like is-abstractions are producing equivalence classes (reflexivity, symmetry, transitivity) of their elements building abstract objects. As-abstractions are multi-dimensional abstractions delivering multi-equivalences and poly-abstract objects. Morphic abstractions are abstracting from any objects retaining only the structure of the multi-dimensional abstract object. As second-order abstractions they are rejecting any kind of *identification* and *reference* to objects.

Morphogrammatics and architectonics

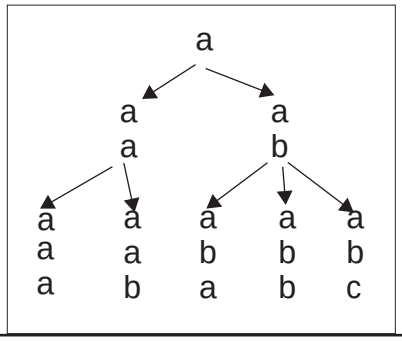
Morphogrammatic operators might be understood as operators able to change the architectonics of a system of programming. Intra-systemic operators are based on their architectonics, thus don't have direct access to transform their underlying architectonics. But polycontextural systems, which are involving the morphic abstraction, too, are capable of transforming their architectonics, thus, producing a kind of self-transformation or self-modification. There is nothing mysterious in changing priorities and paradigms of an evolved re-solution. Suddenly, other aspects which hadn't be prior on the list are emerging in the development and are becoming dominant.

Morphogrammatics at: http://www.thinkartlab.com/pkl/media/mg-book.pdf

5.3 Morphogrammatic evolution of object-schemes

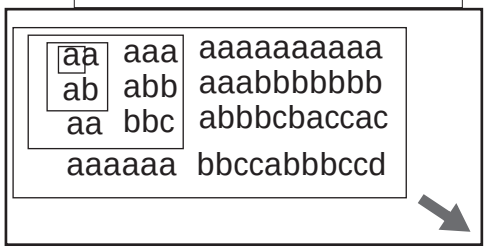
Object-schemes are defined as dynamic patterns. Object-schemes as morphograms can grow and reduce naturally along the rules of emanation and evolution of morphogrammatics. These rules are not so obvious as the composition / decomposition out of elementary parts pre-defined in a repertoire.

Tree representation of morphograms



The evolutionary development of morphograms is determined by its Sterling Numbers of Second Kind and not by a pre-given successor operation. Thus, the quasi dyadic, triadic, n-adic structure of the development tree is determined not by an alphabet nor the successor operation but by the morphograms involved at the place of evolution. That is, the 4th step of the development is determined only by the facts of the 3rd step. And not abstract by an atomistic alphabet and its operators.

Tabular development

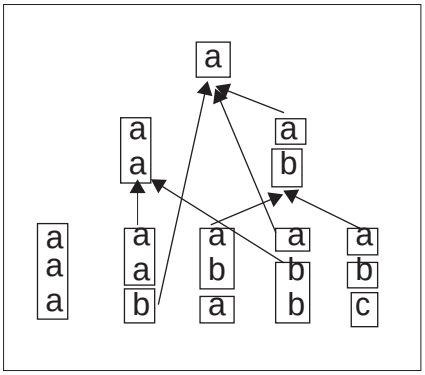


The evolutionary development of morphograms is determined by its Sterling Numbers of the Second Kind and not by a pre-given successor operation, producing a free monoid. Thus, the quasi dyadic, triadic, n-adic structure of the development tree is determined not by an alphabet nor a stable successor operation but only by the just morphograms involved at the place of evolution. That is, the 4th step of the development is determined only by the facts of the 3rd step. And not in an abstract way by an atomistic alphabet and its operators.

5.4 Morphogrammatic devolution of object-schemes

The holistic characteristics of morphograms becomes even more obvious if we observe their decomposition (devolution) into parts.

Graph of monomorphies



Decomposition of complex object-schemes, morphograms, is, in accordance to its evolution, not a devolution into its elementary or atomistic parts, but into its *monomorphies*. Monomorphies are the parts (Teile) of a whole (Ganzheit).

The diagram gives an idea of devolution of the 5 morphograms into monomorphies.

An atomic understanding of morphograms, say as sequences of signs, would decompose the morphogram [aaa] into {a,a,a}, thus into {a}. But considered as a morphogram, [aaa] is not a sequence but a pattern, a Gestalt (morphe), of homogeneous structure and therefore not accessible to decomposition into morphic parts.

This may hint to the fact that evolution (composition) and devolution (decomposition) of morphograms are not ruled by a simple word algebra which is producing an economy of linear sequences of signs.

5.5 Morphogrammatic operations on object-schemes

There are a lot of interesting operations on morphograms. To give an idea I mention only a few. Additional to the operations of evolution / devolution and reflection different kinds of fusions and de-fusions are possible. A kind of concatenation called @-fusion (Verkettung, Vk) may be the most familiar. Other types of fusion are the operation of *affiliation* (Verschmelzung, Vs) and the operation of *compacting* (Verdichtung). To all these operators the inverse operation has to be introduced, thus to concatenation *de-concatenation* (Ent-Kettung, EVk), to affiliation *de-affiliation* (Ent-schmelzung, EVs), and to compacting *de-compacting* (Ent-Dichtung). On the base of these operations it can be shown that the morphogrammatic equivalence is not depending on the "length" of compared morphograms. Thus, *Two morphograms are morphogrammatic equivalent iff they can be decomposed into equal monomorphies.*

The following examples may illustrate the general idea of fusion / de-fusion.

5.5.1 @-fusion of morphograms

$$@-\ fusion\ of\ 2\ morphograms\ [aab], [ab]:$$

$$[aab]\,@[ab] = \begin{Bmatrix}[aabab], [aabba], [aabac], [aabbc] \\ [aabca], [aabcb], [aabcd]. \end{Bmatrix}$$

The @-fusion can be compared to the semiotic concatenation but not accepting the identity of signs. It can be seen as a kind of a "domino" continuation.

5.5.2 &-fusion of morphograms

$$\&-\ fusion\ of\ 2\ morphograms\ [aab], [ab]:$$

$$[aab]\,\&[ab] = \{[aaba],[aabc]\}$$

The &-fusion is connecting two morphograms by "melting" on one kenogram.

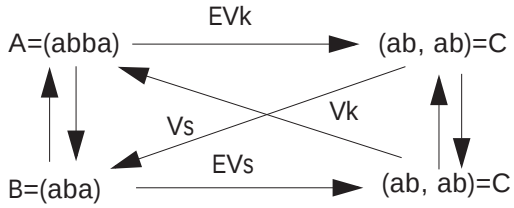
5.5.3 $-fusion of morphograms

$$\$-\ fusion\ of\ 2\ morphograms\ [aab], [ab]:$$

$$[aab]\,\$[ab] = \{[aab]\}$$

The $-fusion is melting morphograms into monomorphies of the same structure.

5.5.4 Morphogrammatic equivalence of [abba] and [aba]



Also length([abba]) > length([aba]),

mg-equivalent([abba], [aba]) iff EVk * Vs = EVs * Vk.

This morphogrammatic equivalence can be compared with the co-algebraic concept of *bisimulation*. Two morphograms are equivalent iff they *behave* the same. This observation maybe the most radical departure from a semiotic understanding of writing.

5.6 AOP-strategies onto Morphograms

The idea of object-schemes, understood as *morphograms*, is enabling a further step of conceptual concretization of the approach of multi- and poly-paradigm programming. Additional to the 2 dimension of *reflectionality* and *interactionality*, as developed earlier, a kind of qualitative differences are involved into the complex distributions. It is important to understand that in a classic approach of programming, multi- or uni-paradigmatic, diversity is introduced by types, sorts, structures, etc. all having a clear semantic characterization and are ruled by a single underlying logic (FOL).

Morphograms in contrast are introducing diversity on a pre-semantic level, considering only the *localization*, the placing of paradigms in the polycontextural grid and not their semantic characteristics. But to differentiate paradigm-schemes or patterns of paradigms from each other they have to be able to be distinguished. And because there are no data types, objects, involved, it can not happen on the usual level of abstraction of identity / diversity. This kind of distinction, thus, is localized on a morphogrammatic level, established by the *morphic abstraction*, delivering differences beyond semiotic and logical principles of identity and identification.

> "Das polykontexturale Objekt nimmt auf Grund seiner internen Komplexität nicht einen, sondern mehrere Orte simultan ein, es ist also polylokal. Das reine poly–lokale Objekt in Absehung jeder kontextur–logischer Thematisierung, bezogen nur auf seine Architektur bzw. Komplexität seiner Substanz, als reines Dies–da, ist bestimmt allein durch die Struktur seiner Örtlichkeit, und diese wird notiert in der Kenogrammatik als *MORPHOGRAMM*."
> OVVS 1985

The morphic abstraction can be understood as the process (subversion) of unveiling the invariant patterns of disseminated contextures containing different programming paradigms. Patterns, neutral and invariant to the viewpoints which are behind object schemes.

Thus, in a simple wording, to apply, say 5 times the same programming paradigm from 5 different points of view to deal with 5 different aspects would be invariant in respect of the chosen paradigm. It wouldn't make a difference which paradigm is applied, only the fact that there are 5 times the same is of interest. If there would be 5 different programming paradigms for 5 different aspects, a permutation of the 5 paradigm wouldn't make a difference under the morphogrammatic abstraction. But obviously, the two examples are morphogrammatically different, [aaaaa] ≠ [abcde]. On the base of morphogrammatic equivalence between programming paradigm patterns translations from one realization to another are well founded. Applying the transformational rules of morphogrammatics to paradigm patterns interesting evolutions / devolutions and metamorphosis are accessible to programming. Conflictive constellations between different approaches are more directly analyzed on a morphogrammatic level than through the disturbing complexity of conflicting programming paradigms.

Also the morphogrammatic abstraction is introducing a highly abstract level of understanding programming, it has a great concrete importance if we consider, in addition to the concepts of AOP, the complex and dynamic *architectonics* of distributed and mediated programming paradigms.

The study of object-schemes, morphograms of distributed programming paradigms and architectonics of programming systems as sketched in *ConTeXtures* could enhance the computation of many so called intensional, metaphoric and complex conceptual constellations which occur, e.g, in reflective security systems.

6 AOP-style: A multi-paradigm view?

The Panalogy Principle: *If you 'understand' something in only one way then you scarcely understand it at all—because when something goes wrong, you'll have no place to go.* Marvin Minsky

6.1 AOP-Strategies

From a systematic point of view there still exists the idea in AOP-programming that a nice hierarchy between the main systematic "components" or programming paradigms has to rule the business. First *Methods*, second *Objects*, third *Aspects*.

Aspect-oriented programming (AOP) grew out of a recognition that typical programs often exhibit behavior that does not fit naturally into a single program module, or even several closely related program modules. Aspect pioneers termed this type of behavior *crosscutting* because it cut across the typical divisions of responsibility in a given programming model. In object-oriented programming, for instance, the natural unit of modularity is the class, and a crosscutting concern is a concern that spans multiple classes. Typical crosscutting concerns include logging, context-sensitive error handling, performance optimization, and design patterns.

AOP introduces aspects, which encapsulate behaviors that affect multiple classes into reusable modules.
AOP addresses a problem space that object-oriented and other procedural languages have never been able to deal with.

AOP complements object-oriented programming by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit. These units are termed aspects, hence the name aspect-oriented programming. By compartmentalizing aspect code, crosscutting concerns become easy to deal with.
Aspects of a system can be changed, inserted or removed at compile time, and even reused.

ftp://www6.software.ibm.com/software/developer/library/j-aspectj.pdf

OOP-object hierarchy

"A language will be called object-oriented if it is object-based and additionally requires that objects have classes and classes have inheritance:
object-oriented = objects + object classes + class inheritance." Peter Wegner

"Data abstraction
Objects should be described as implementations of data types." Bertrand Meyer

"public class *Object*
Class *Object* is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class." Java

AOP-aspect heterarchy?

On the other hand it seems that AOP starts to think and act more *heterarchically* than allowed by its systematics. With mechanisms such as *crosscutting concerns* (logging, security, persistence) and then *weaving* parts together, elements of horizontal and combining strategies enter the game and are looking for adaquate implementation.

"Areas organized around classes of parts of systems are called horizontal domains*. Obviously, specific systems or components within a domain share many characteristics because they share many requirements."* Generative Programming, p. 20

For AOP, *aspects* are basic, they contain *pointcuts* (set of joinpoints) and *advices*. Their role is systematically similar to *objects* or *class* for OOP, they are basic. Again, a unifying concept, aspects, is governing the hierarchy of the system.

AOP- hierarchy/heterarchy interaction?

The mixture of approaches of AOP is designing a tabular structure of horizontally distributed aspects and vertically distributed concerns. That is, AOP is located in a matrix of an interlocking mechanism of *hierarchic* and *heterarchic* dimensions.

This dynamics is well described by Czarnecki/Eisennecker in GP as follows:

"In a AOP system, components and aspects are woven *together to obtain a system implementation that contains an intertwined mixture of aspects and components."*

"A model is an aspect *of another model if it crosscuts its structure."* This connection has not to be stable *"Because an aspect is relative to some model, it might be possible to refactor the model so that the aspect ceases to be an aspect of the model."* This flexibility has to be handled. *"One way to deal with this problem is to introduce a mediator component that encapsulates these interaction patterns."* But such kind of mediator are working like buffers, they are not proper parts of the conception of interaction. Also the "chiastic" exchange aspect/component is not modeled by an intrinsic mechanism of the general programming paradigm. It would contradict its hierarchic order. *"Thus, we can refactor our design and turn an aspect into a component."* GP, p. 265/66

Why all the fuss?

If we consider paradigms in the multi-paradigm approach of AOP simply as intra-contextural applications of the as-abstraction, that is, only as contextually different paradigms, then the multitude of its paradigm can always be *reduced*–without serious loss–to a uni-paradigm solution. Multitude then is simply a question of tradition, taste and some economy. Lacking any strict conceptual foundation. Thus, what you can do with AOP you can do it also with, say OOP. What programmers anyway always did.

Nevertheless, there is always some advantage to thematize and focus on everyday practice and to bring it to an explicit and systematic treatment. But if you want more, you may have to swallow the pill.

6.2 Multitudes, simultaneity and profundity

6.2.1 Multi-paradigm view

> *"Thus, the impact of AOP on modeling methods will be fundamental. In order to understand this impact, we have to realize that most of the current analysis, design, and implementation methods are centered around single paradigms, for example, OO methods around objects and structured methods around procedures and data structures. The insight of AOP is that we need different paradigms for different aspects we want to build. This kind of thinking is referred to as the* multiparadigm view*."* GP, p. 331

There are not only different aspects to consider, which could be dealt by a common programming paradigm, but also different paradigms for different aspects may be involved. Because there is no programming paradigm which suits for all tasks in the same way, different paradigms have to be put on scene to reach an optimal performance. This multi-paradigm approach is strengthening the AOP position towards more flexibility and complexity. Also not mentioned, it seems to be obvious, that those different paradigms are not applied only successively but *simultaneously*. Hard problems of mediating, combining and weaving different approaches into a successful play are arising. Programming dynamic complexity isn't easy and AOP is not yet delivering strong enough methods to deal with it. As long as aspects are simply build on top of OOP there is not much chance for a radical revision and enlargement of the core language.

It seems that the *multi-paradigm* view as applied today, say by Ruby, Python, O'Caml or ARS, is mainly a methodological way of viewing in different ways at problem solving methods and is not incorporating in itself a multiplicity of paradigms into the basic definitions of the language. The methodological approach is more a question of programming *styles* and is not touching the very structure of the language itself. In this sense you can program FORTRAN-like in JAVA and OOP-like in LISP, and so on. Incorporating them all together in a family of programming styles.

Multi-paradigm vs. Poly-paradigm

Thus, we can distinct between multi-paradigm view as a successive application of different paradigms and the more complex, multi-paradigm view as a simultaneous application of different paradigms playing together to deal with a complex constellation. Simultaneity of the application means that the complex constellation, problem complexion, has to be considered at once in the light of different programming styles. A complex situation has to be described by complementary statements and not by a summation of different paradigmatic statements. Thus, it may be necessary that a complex constellation is dealt at once, complementary, with an object-oriented approach and a functional and an aspect-oriented and a XY-approach. Such a focussation of a problem space is not only multi-paradigmatic but also multi-perspective highlighting the complexion in the spectre of the light. This kind of simultaneous multitudes of paradigms should be called *poly-paradigm. Poly-paradigm* is possible only if such a plurality is incorporated into the very basic features of the architectonics of the complexion of programming language(s). Thus, multi-paradigms are defined intra-contextural, each contexture can realize its own multi-paradigm, while the poly-paradigm approach requires mediated contextures as realized in poly-contextural systems. Poly-paradigm in programming is thus not only a multitude of paradigmatic styles and approaches a programmer can choose but part of the very architectonics of a complex programming system. Multi-paradigmatic approaches are not dealing properly with complex situations, they thematize them still as simple systems (Robert Rosen).

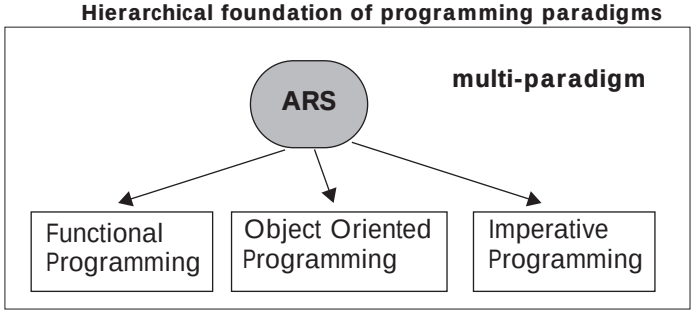
6.2.2 Diagrams of multi- and poly-paradigms approaches

To visualize the idea of poly-paradigmatic programming in contrast to multi-programming some diagrams may be introduced.

6.2.2.1 Multi-paradigm view as proposed by ARS.

"*A multiparadigm programming language* is a programming language that supports more than one programming paradigm."
http://en.wikipedia.org/wiki/Multi-paradigm_programming_language
http://en.wikipedia.org/wiki/Categorical_list_of_programming_languages
http://www.dmst.aueb.gr/dds/pubs/thesis/PhD/html/thesis.pdf

Hierarchical foundation of programming paradigms

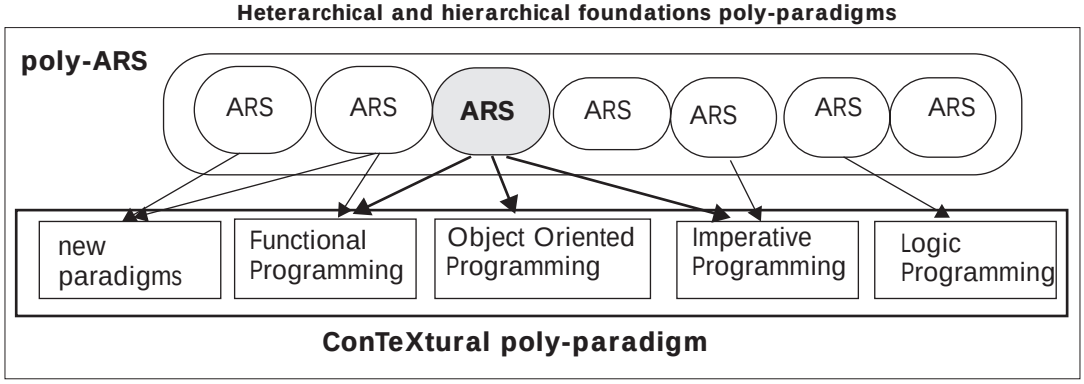


Similar to the Ruby, Python, Oz, OCaml and other modern programming languages ARS (Georg Loczewski) is proposing, and as far as I understand, founding and realizing, a real multi-paradigm approach from its very basic definition which is a kind of a strict Lambda Calculus interpretation as its *kernel language*. All basic components of, say functional, object-oriented and imperative programming styles can be defined on the base of ARS (Abstraction+Reference+Synthesis). But obviously, they can't be applied at once to a complex object (situation) simply because there are no such monsters implied or involved in the game (of the ultimate lambda calculus). Nowadays, it is becoming quite fashionable to christen such multi-approaches as polycontextural. But this is only name-dropping and confusion, a renaming, say of multi-perspectivism.

http://www.sics.se/%7Eseif/Publications/fdp.pdf
http://www2.info.ucl.ac.be/people/PVR/BCStalk.pdf

6.2.2.2 Poly-paradigm scenario as proposed by ConTeXtures

Heterarchical and hierarchical foundations poly-paradigms



Poly-paradigms are based in polycontextural programming languages where multitude and complexity is incorporated in the very axioms of the design of the language. Such an approach is not only poly-contextural but also *dis-contextural* considering the abyss between contextures and at once *trans-contextural* in the sense of mediating those different contextures together to an interplaying complexion.

6.2.3 Hierarchies are not enough

A key intuition underlying our work is that simple hierarchies are not rich enough to capture complex structures. In fact, we believe that any single ontological structure is insufficient. As a result, we have been exploring a variety of mechanisms that make it possible to view implement a system from multiple perspectives. One thing we believe about these multiple perspectives is that in order to have significant power they must be able to crosscut each other. That is, one perspective must be able to organize the implementation in fundamentally different ways than other perspectives.

The thread that has been common to our projects is rooted in the fact that the normal rules of modularity do not suffice for describing quality software.

http://www2.parc.com/csl/groups/sda/

Simultaneity of paradigms and multiple perspectives are asked if those perspectives should be able to *crosscut* each other in a serious way. But this exactly isn't possible in a hierarchic setting of conceptualization, modeling and programming.

6.2.3.1 Some more readings: Hyperspaces (H. Ossher and P. Tarr)

most powerful abstraction

Is AspectJ worth using? Grady Booch describes aspect-oriented programming as one of three movements that collectively mark the beginning of a fundamental shift in the way software is designed and written. (See his "Through the Looking Glass" in the Resources section.) I agree with him. AOP addresses a problem space that object-oriented and other procedural languages have never been able to deal with. Within a few weeks of my introduction to AspectJ, I've seen it provide elegant, reusable solutions to problems I thought were fundamental limitations of programming. It's fair to say that AOP is the most powerful abstraction I've learned of since I began using objects.

Of course, AspectJ does have a learning curve. As with any language or language extension, it has subtleties that you need to grasp before you can leverage its full power. However, the learning curve is not too steep -- after reading through the developer's guide and working through a few examples, I found myself ready to compose useful aspects. AspectJ feels natural, as if it fills in a gap in your programming knowledge rather than extending it in a new direction. Some of the AspectJ tools are a bit rough around the edges, but I haven't encountered any major problems.

modularize the un-modularizable

Given the power of AspectJ to modularize the un-modularizable, I think it's worth using immediately. If your project or your company aren't ready to use AspectJ in production, you can easily apply AspectJ to development concerns such as debugging and contract enforcement. Do yourself a favor and check out this language extension.
http://www-128.ibm.com/developerworks/library/j-aspectj/

patterns

Patterns are indeed a sign of a break from the traditional von Neumann model of computation, because they name things that are somewhat orthogonal to the basic mappings of code to executables we have in our heads.
Second, there's the growing understanding of the importance of multiple views in the science and practice of software architecture

simultaneously–no tyrant

It is necessary for developers to be able to identify and encapsulate any kinds, or dimensions, of concern, simultaneously. Further, all dimensions must be created equal—there must not be “tyrant” dimensions that preclude decomposition along other dimensions.
http://www.research.ibm.com/hyperspace/Papers/index.htm

multi-dimensional separation of concerns

These goals, while laudable, have not yet been achieved in practice. We believe this is because the set of relevant concerns varies over time and is context-sensitive--different development activities, stages of the software lifecycle, developers, and roles often involve concerns of dramatically different kinds. Thus, any criterion for decomposition will be appropriate for some contexts, but not for all. Further, multiple kinds of concerns may be relevant simultaneously, and they may overlap and interact, as features and classes do. We use the term multi-dimensional separation of concerns (MDSOC) to refer to flexible and incremental separation, modularization, and integration of software artifacts based on any number of concerns. It overcomes limitations of existing mechanisms by permitting clean separation of multiple, potentially overlapping and interacting concerns simultaneously, with support for on-demand remodularization to encapsulate new concerns at any time.
http://www.research.ibm.com/hyperspace/

overlapping or interacting concerns

We use the term multi-dimensional separation of concerns to denote separation of concerns involving:

* Multiple, arbitrary kinds (dimensions) of concerns.

* Separation according to these concerns simultaneously; i.e., a developer is not forced to choose a small number (usually one) of "dominant" dimensions of concern according to which to decompose a system at the expense of others. This separation must be more than just identification of concerns and of the code that pertains to each. It must include segregation (encapsulation) that is sufficient to limit significantly the impact of change. This does not mean that every change within a concern can affect only that concern; this is never possible. It means that, just as modules in a dominant decomposition localize the impact of many kinds of change and limit the impact (propagation) of others, so must this be true of any of the concerns. More precise definition of this requirement is an interesting topic for discussion and for further research.

* Overlapping or interacting concerns. It is appealing to think of many concerns as being independent or "orthogonal," but this is rarely the case in practice. It is essential to be able to support interacting concerns, while still achieving useful separation.

http://www.research.ibm.com/hyperspace/MDSOC.htm

no new languages

Realizations of MDSOC can permit incremental identification and encapsulation of concerns, without requiring the use of new languages or formalisms.
http://www.research.ibm.com/hyperspace/index.htm

Myth 2: AOP doesn't solve any new problems

Reality: You're right -- it doesn't!
This is one AOP "myth" that is actually correct. AOP isn't a new computation theory that solves yet-unsolved problems. It's merely a programming technique that targets a specific problem -- modularization of crosscutting concerns.
http://www.nofluffjuststuff.com/speaker_view.jsp?speakerId=8

Blog Aspectivity

Blog Aspectivity , Ramnivas Laddad
http://ramnivas.com/blog/index.php?p=22

downloads
http://www.eclipse.org/aspectj/

6.2.4 AOP and ConTeXtures

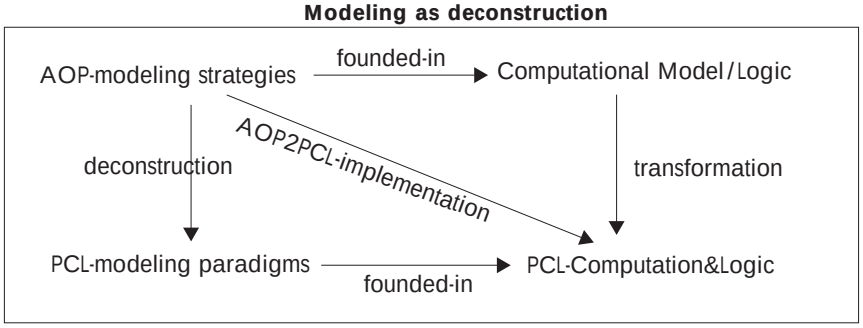
New approaches, like combining logics, fibred logics, polycontextural logics, all dealing in different ways with problems and techniques of handling complex systems are not yet recognized and applied to modern programming and conceptual language development.

With that, we are close to the polycontextural and multi-paradigm approach of ConTeXtures and the question of mapping AOP strategies onto the polycontextural matrix (PM) of ConTeXtures arises. What is called *"intertwined mixture"* or *"weaving"* and "*combining*" may have a correspondence and a further explication in the concept of *mediation* in polycontextural logics (PolyLogics).

Nevertheless, the crucial difference then between the design of AOP and ConTeXtures is this. AOP is described as a *modeling* method based on classic programming languages while ConTeXtures aims to implement a multitude of paradigms and viewpoints into the very basic constructs of its programming language as such. Thus, dynamic complexity in ConTeXtures is involved at the very beginning of its core language and not as a late demand from complex concerns.

AOP is using meta-programming techniques but is based on a core language which is excluding self-referential structures which would be basic for extensive reflectional and meta-programming.

Modeling as deconstruction



Because new ideas don't fall from the sky, polycontextural strategies have to be developed in a process of contrasting with existing new trends in logic and programming developments.

The diagram should prevent decisively enough any confusions between the 4 domains and their relations under consideration. The use of classic methods to explain and formalize polycontexturality is not denying some abuse of terminology, but is insisting on its clear positioning and its awareness of use/abuse strategies. Thus, the study as use/abuse of involved concepts develops a certain kind of *deconstruction* of modern programming paradigms with the aim to elaborate a way for an introduction of polycontextural approaches to programming.

Obviously, the common terms are *plurality*, *heterarchy*, *multitudes*, *mediation* all being used in a way by the introduction of new programming paradigms which allows to separate some strata from their traditional understanding, which are in strict conceptual conflict with the grounding postulates and requirements of the approaches. Those strata, based on *analogy*/*polysemy*, and their *de-sedimentation*, are legitimating a deconstructive reading and transformation towards a graphematic paradigm of programming.

http://www.thinkartlab.com/pkl/lola/ConTeXtures.pdf

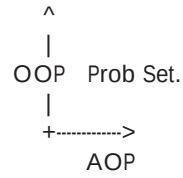
Feature based approach: http://www.cis.upenn.edu/~bcpierce/papers/tt-jfp.pdf

7 AOP-strategies onto PM

The programming paradigm of ConTeXtures lives from the "2-dimensional" distinction of *reflectional* and *interactional* strategies of computational processes. Both together are implemented in the polycontextural matrix (PM). Thus, it seems to be quite natural to model and implement the 2-dimensional approach of AOP into the framework of ConTeXtures. Additional "patches" to existing programming paradigms are not doing the job properly because they are nevertheless based on a "one-dimensional" programming approach lacking any real orthogonality/transversality.

7.1 Overview of AOP

"In AOP, one considers aspects of concern applicable across multiple classes and methods. Thus AOP is said to address cross-cutting concerns. Aspects consist of advice, which are methods designed to intercept other methods or events according to specified criteria. This criteria is called a point-cut and it designates a set of join-points. A join-point (or code-point) is the specific place within a program's execution where the advice can be inserted. In this way, AOP is thought to provide a means of organizing code ***orthogonal*** *to OOP techniques.*



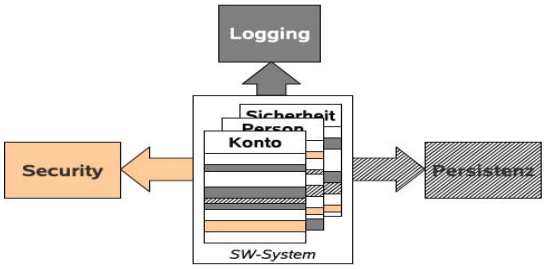
Since AOP is a very powerful paradigm for abstracting programming solutions into separate concerns, and shows great promise for improvements in code maintenance and reusability, it seems only natural that an agile language such as Ruby would provide support for this increasing popular pattern of design."
http://www.rubygarden.org/ruby?AspectOrientedRuby

Cutting, splitting and *weaving* needs conceptual space, a kind of a topology, to be realized, otherwise it is producing paradoxes or it will be reduced to triviality. OOP is not delivering necessary range because of its strict hierarchic main structure. To put aspect-oriented strategies onto OOP would need an additional dimension to allow *orthogonal* and *transversal* conceptualization and realization. Without a generalizing approach, the aspects are badly "glued" onto the traditional hierarchy of OOP. Thus, to be able to work, OOPxAOP should be more than AOP or aspect-augmented OOP.

"The problem of splitting what is intentionally supposed to be a single object is referred as object schizophrenia*. [...] Another problem results from the fact that we have to manage two object identities instead of one."* GP, p. 294. Obviously, an object is an object in this way of thinking. Any splitting or double use of it is generating ontological problems. But an object in the schemes of as-abstractions, i.e. thematizations, is "naturally" be cloned and duplicated at once into other objects. Not falling apart, but being involved in the game of mediation, not ruled (out) by the master class Object.

In contrast: *"The first real step in implementation is, of course, the creation of the transparent subclass, the Cut. This requires an addition in the structure of an object's class hierarchy;[...]."*
"After I had looked at AspectJ, I had found that aspects, although an interesting idea, seemed to break object-orientation rather than enhance it. This implementation, on the other hand, is strongly based on OOP, and thus works very well with it!"
http://www.rcrchive.net/rcr/show/321

7.2 A general sketch of mapping AOP onto PM



This diagram shows a bank account object with (account, person, security) embedded in horizontal concerns, aspects of logging, general security, persistence in a software system. It turns out that the handling of the environment takes a great deal of interest and maintenance, while the business program is of quite simple structure. But this static situation turns into an opposite picture if interests are focussed on the environment, say security. Then, the security concerns are primary and the business application becomes secondary. AOP is not offering any mechanism to deal with dynamics of this kind.

The idea behind a mapping of AOP or OOP onto the proemial matrix PM is inspired by the possibility of a chiastic dynamization of the main concepts of AOP in the play and their distribution along the dimension of *reflectionality* and *interactionality* of mediated systems. Thus, a *heterarchic cut* is deliberating the main concepts from their hierarchy and is involving them into chiastic interplay. Between objects, classes, aspects, domains, viewpoints, etc., hierarchies are established only as temporary frozen chiasms. The whole game is played as an interplay of heterarchy and hierarchy.

7.2.1 Parallel computations of singular mediated systems

The diagram shows 3 parallel distributed and mediated programming systems.

O_1			O_2			O_3			$PM^{(3)}$
M1	M2	M3	M1	M2	M3	M1	M2	M3	$\begin{bmatrix} O1 \\ \begin{pmatrix} M1 M2 M3 \\ (S100) \end{pmatrix} \end{bmatrix}$
↓	#	#	#	↓	#	#	#	↓	$\begin{bmatrix} O2 \\ \begin{pmatrix} M1 M2 M3 \\ (S020) \end{pmatrix} \end{bmatrix}$
↓				↓				↓	$\begin{bmatrix} O3 \\ \begin{pmatrix} M1 M2 M3 \\ (S003) \end{pmatrix} \end{bmatrix}$
↓				↓				↓	
S_{100}			S_{020}			S_{003}			

In this case, each system works isolated at OiMi without interaction with its neighbor systems or reflection into itself. Thus, it is pictured as the diagonal of the matrix PM.

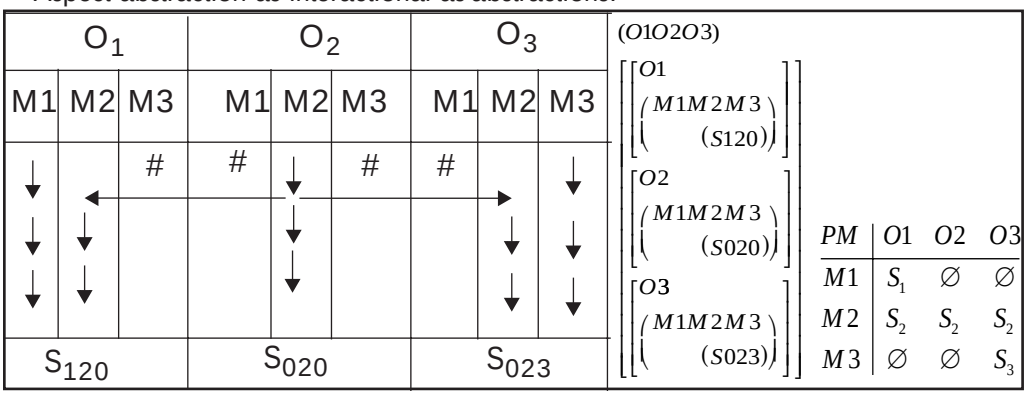
PM	$O1$	$O2$	$O3$
$M1$	S_1	$\varnothing$	$\varnothing$
$M2$	$\varnothing$	S_2	$\varnothing$
$M3$	$\varnothing$	$\varnothing$	S_3

PM consists of loci which can be occupied by systems. In this case, the dimensionality of PM is restricted to two, reflectionality and interactionality. Each system S in PM can be a full programming paradigm (language) like OOP or AOP. Thus its conceptual model consists of 3 parallel mediated hierarchies, represented as 3 mediated trees.

As an *example*, each concern is focussed separately and has its own representation. The business concern is treated separately, also the security concerns and the persistency. The as-abstraction scheme, therefore, is: Security is security, bank account is bank account, logging is logging and persistency is persistency. But those self-identical features are, nevertheless, mediated together in a complexion represented as the whole system.

7.2.2 Interactional situation

Aspect abstraction as interactional as-abstractions.



The reflectional situation is considering an object as an object in the sense of the as-abstraction "contexture$_i$ as contexture$_j$ in a poly-contexture".

Thus, the security aspect of the business aspect is treated while simultaneously the security as security and the business as business aspect is under focus.

Co-operation: Interactions as commands

In the example there is an addressing from O2 to O1 and O3 realizing positioning at once in O1 as O1M2 and in O3 as O3M2 while keeping O2M2 as the addresser of O1 and O2 persistent. The same simultaneous autonomy of agents holds for O1 and O3. The complexity of the agents O1 and O3 are giving space to the possibility of acceptance of the interaction of O2. Thus, it is not O1 as O1M1 itself which is accepting the addressing from O2 but O1M2 as a model of the interaction from O2 to O1. In the diagram the mutual interaction of acceptance and rejection has to be interpreted, and is not visualized as such. That is, the action of O2M2 to O1M2 means, that O2 is rejecting the job and addressing it to O1 and O1 mutually accepts the job in offering computational space to O2 at O1M2.

The realization of an interaction, as described, can be understood as a co-operative command. An agent or processor O2 is commanding on the base of realized interactivity with agent O1 and O3, that is, of realized architectonics, to proceed a task of O2M2 at O1M2 and the same with O3 to proceed at O3 a task of M2 at O3M2.

Traversal strategies and addressing

Interactional situations may give a hint how to deal in a polycontextural programming environment with *traversal strategies*. Maybe surpassing the *Law of Demeter*, concerning "information-passing", more flexible strategies of interaction are necessary.

Interactivity is not based on communication as information processing or message passing in hierarchic systems. It can be described as the action of addressing an addressee which is able to accept the addressing by its own addressable structure. After having been addressed and the addressing is accepted by the addressed and the addresser has recognized the acceptance of being addressed and the addressing is thus established, information can be exchanged between agents in the sense of processual communication. Thus, on the base of realized interactivity the informational aspects of tasks can be considered. Only on the base of this interactional agreement information exchange can happen. Obviously, this addressing activity is creating its own interactional topology and is not depending on a pre-given hierarchic object domain.

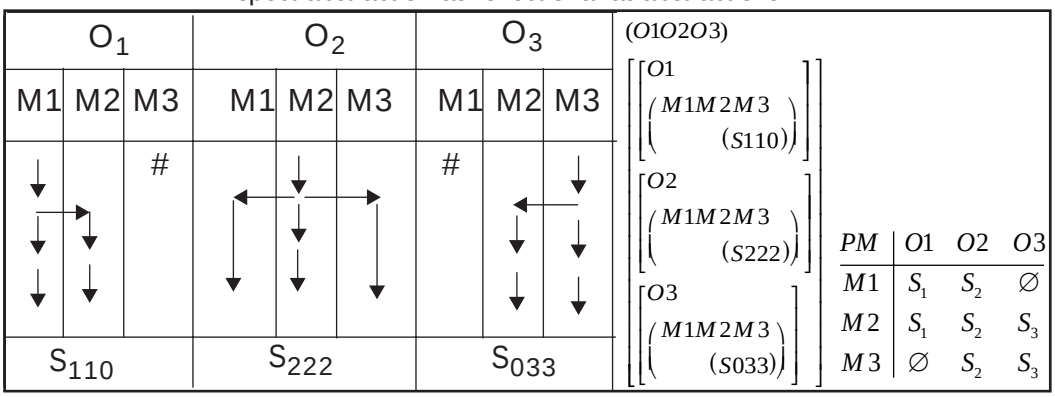
Acceptance and rejection

Therefore, the structure of interaction is always complex: at once realizing the addresser and the inner environment of the addressee. This simultaneity of inner and outer environments of agents is involving a kind of structural bifurcation and a mutual actions of *acceptance* and/or *rejection* of the involved agents based on complexity of their architectonics. That is, the addressee has to give space (einräumen) to the addresser to be addressed. To address and to accept to be addressed is a *mutual* action of at least two agents in a common co-created environment. Self-addressing would be a case of interactional reflectionality of an agent into itself. Interactivity therefore is a mutual action of *acceptance* and *rejection* between different agents.

This scenario of acceptance and rejection has an operative representation in the poly-logical laws of transjunctions. *Transjunctions* and *multi-negations* are the basic operations of polycontextural logics. They are guiding the design of polycontextural programming languages, too.

7.2.3 Reflectional situation

Aspect abstraction as reflectional as-abstractions.



Reflectional situations are not modeled in classic conceptual modeling. A concept is a concept. It makes no sense to construct the concept of a concept. It simply would be a concept, again. But, say, the 'security of security' is not only a possible reflection, but a crucial question for real security systems.

The diagram shows a single reflection of system1into itself, a two-fold reflection in itself of system2 and a single self-reflection of system3. Reflections as iterations or replications of objects need their own place to be implemented, embedded and realized. Thus, the matrix PM is offering the iteration of O1M1 at locus O1M2. Similar, O2M2 is re-placed at O2M1 and at O2M3, and O3M3 at O3M2.

Reflections as simple iterations without re-placement of the reflected system are nevertheless not excluded at each position of the constellation.

This reflectional situation is considering an object as an object in the sense of the as-abstraction *"name$_i$ as name$_j$ in a contexture".*

Contextu(r)al modules

Reflections on security, security as security, meta-reflections on security, all need their own loci to be realized. All main objects or agents of a system should be chiastified to have full transparency and access to reusability of *contextu(r)al modules.*

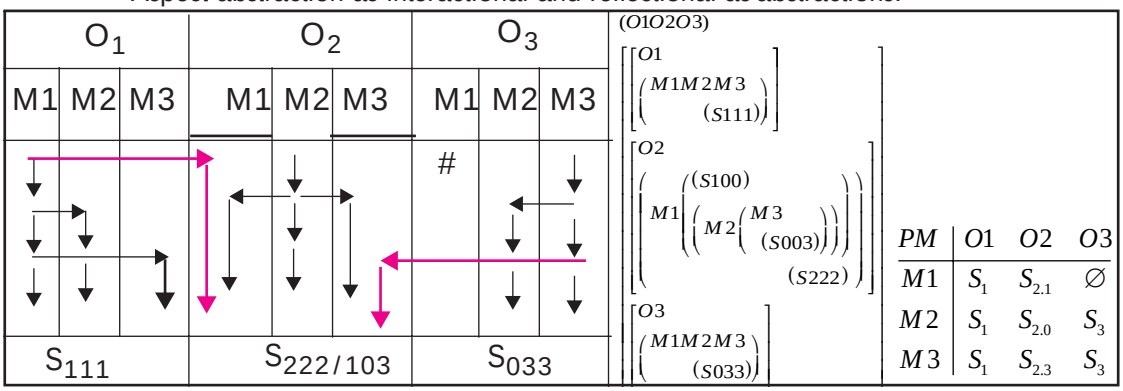
Reflection: Security aspects of security; business aspect of business; etc.

Interaction: Business aspects of security; security aspect of business; etc.

Reflection+Interaction: Security aspect of security and business and persistency.

7.2.4 Interplay between interactional and reflectional situations

Aspect abstraction as interactional and reflectional as-abstractions.



Even for a system of small and stable complexity highly complicated constellations of intertwined interactionality and reflectionality are accessible. The diagrams, matrices and bracket formulas are not presenting a calculus but are only simple tools to explain and demonstrate the main idea of polycontextural modeling and computing.

Conceptual analysis shouldn't restrict itself to hierarchy producing is-abstractions.

To introduce different kinds of as-abstractions goes hand in hand with the introduction of new types of inheritance rules not only between objects, components, modules, contexts but also between contextures, i.e., programming paradigms. Thus, aspects can be considered as contextures and contextures can be realized as aspects. Reflectionality and interactionality are, among others, two strong new kind of *meta-programming* introduced at the very score of the programming languages. Polycontextural programming is not primarily problem solving, exposed by methods and objects, but reality construction involving the contextural dynamics of systemic points of view.

Modifying the computational system

Interactional and reflectional actions are augmenting the environmental space of the computational systems considered as main systems. In the case of the examples, the diagonal systems. What is a main system is depending on decisions and can change into other modi of the environment. On the other hand, it is not excluded that results from those environments will have the possibility to modify the computational systems. Such a modification would then be mediated through the experiences of reflection and interaction with the system itself and with its neighbor systems.

Again, all the proposed strategies are structural and functional conceptual considerations, prior to any information processing. Thus, in no way to be understood as *cybernetic* information processes. Information processing then is based on structural, i.e., architectonic constructions.

Open to reduction?

The main question remains: *Is the enlarged system a conservative extension possible to reduction?* Conservative extensions are helpful in many ways but this is not changing anything in their definition to be reducible. A good example is given by the programming language ARS (Abstraction+Reference+Synthesis) in which all known programming paradigms (styles) can be defined. Thus, say OOP, can be reduced, in principle, also surely not practically, to ARS, which is a kind of a generalized lambda calculus.

8 Abjects: Weaving and mediation in a polycontextural setting

A simple example may concretize a little more the idea of disseminated aspects and their interactional and reflectional activities.

> *"Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns."*
> http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

> *"The join-point is central to AOP. It is a specific place in program execution where other code, i.e. advices, may be "inserted"."*
> http://www.rubygarden.org/ruby?AspectOrientedRuby/NotesRoughDrafts

The advice programming text and the aspect programming text are brought together by the operation of weaving producing the final program text. The aspects are composed by program texts distributed over different pointcuts involving joinpoints. Thus, cuts are delivering the texts to the *weaver* to be woven with the advice text to produce the full program.

At a first glance, the polycontextural approach is simply offering a tabular structure to place and localize different program texts not accessible in existing languages. But this is probably not simply a didactical offer but will have systematic consequences for the whole programming paradigm. The main difference between OOP, AOP and the idea of polycontextural programming as proposed in ConTeXture may first be seen in the different use of logic and the command structure of the program. Despite the strong modularization of the program into its aspects (modules) in AOP all aspects are ruled together by one and only one logic and command structure. This monolitic structure is not only running through all aspects and objects but is also guaranteeing the rationality and operationality of the conceptual design of the aspect-oriented program.

Weaving vs. Mediation

Thus, weaving in polycontextural systems is in fact *mediation*. Different programming texts are mediated together to build an interactional/reflectional compound system where all components are defined by their own logic and command structure. Those rationalities can even defer from each other by having different kind of logics in use. Weaving, in contrast, is merging the textual parts together into the main programming text which contains the conceptual frame and the general command structures. Mediation, therefore, has not only to weave aspects together but also to organize different logics to build a working cooperative compound system. On this base of mediated logic aspects can then be distributed and mediated in a heterarchic order. Each heterarchic distributed aspect can incorporate all sorts of hierarchic organizations of its intracontextural components.

Objects + Aspects = Abjects

Aspects are modularizing crosscutting concerns, objects common concerns, the mediation of both, aspects and objects, are realized by the interlocking mechanism of *abjects*. Abjects are the 2-dimensional strategies of programming, behaviors and structures, dynamics and concepts, intertwined together in a complementary play *"[…]where identities (subject/object, etc.) do not exist or only barely so—double, fuzzy, heterogeneous, animal, metamorphosed, altered, abject".* (Julia Kristeva)

Saver, but weaker forms of mediation of logical systems than polycontextural mediations are proposed by the projects of *Combining Logics* (Gabbay) and *Logical Fiberings* (Pfalzgraf). Both should be applied to conceptualize programming.

8.1 From Objects to Aspects, Metapattern and Contextures

8.1.1 ARS-OOP approach to bank account

```
[define make – account
 (lambda (balance)
  self)]

[define make – account
 (lambda (balance)
  (define get – balance
   (lambda ()
    balance)))]

[define deposit
 (lambda (account)
  (define balance
   (add balance amount))
  balance)]

[define withdraw
 (lambda (amount)
  (if (gep balance amount)
   (lambda ()
    (define balance
     (sub balance amount))
    balance)
   false))]

[define self
 (lambda (msg)
  (if (equaln msg one)
   balance
   (if (equaln msg two)
    deposit
    (if (equaln msg three)
     withdraw
     (if (equaln msg four)
      print – account
      false))))))]

[define print – account
 (lambda ()
  (ndisp! balance))]
```

http://www.aplusplus.net/bookonl/node142.html

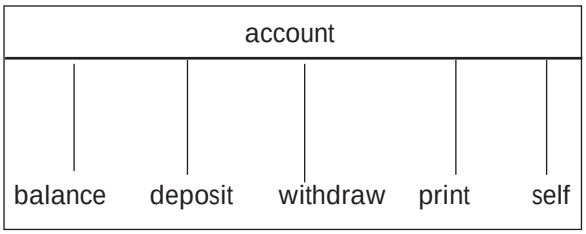
Why should we heterarchize this bank account model?
As we can see, all the modules are quite autonomous defined and embarrassed only by the clam of self which is organizing the modules/objects into a hierarchic order to function as an OO program.
One reason could be the constellation of many different accounts belonging, say to the same user. For each account the particular objects with their specific attributes has to be defined and put together in some kind. The accounts can differ and change over time on all parametric levels: currency, type of printing, etc. In an extreme situation they can even differ in their arithmetics and logic of the accounts, and nevertheless they have to cooperate and work as a complex account.
In a heterachic setting, new features and new modules can easily added and old modules can put to sleep without costing the system anything.
This can be done more efficiently than in re-programming the users accounts into one big new OO constellation. Anytime the user is changing the attributes or the number of accounts this procedure has to be repeated. This kind of flexibility of modularization surely was a main attempt of OOP.
ConTeXtures differs from OOP in choosing additionally to the hierachic organization of the objects the new organizational dimension of heterarchy which deliberates the modules from their hierarchic encloser and offers space for more autonomy and interactivity with the involvement of contextures.
In other words: the root or self is simply an object next to other objects.
Ontologically speaking this says that the whole is taken as a part in another contexture.
The connection between different objects in different contextures is not a link but an interaction between contextures. Even a single class, here account, can be put into a heterarchic setting reflecting its intrinsic structure. (DERRIDA'S MACHINES 2003)

8.2 Metapattern approach

The metapattern approach was introduced by Pieter Wisse to dynamize object-oriented programming. His *situations* are producing *aspects.*

> *"Compared to an object that (only) exists absolutely, an object believed to exist in a multitude of different situations can unambiguously be modeled – to be equipped – with corresponding behavioral multiplicity. […] The radical conclusion from the orientation at situational behavior is that an object's identification is behaviorally meaningless. […] Traditional object orientation assigns identity at the level of overall objects. Context orientation replaces this view of singular objects with that of plurality within the object; the object always needs a context to uniquely identify the relevant part of an overall object, which is what identifying nodes regulate. When behaviors are identical, no distinction between contexts is necessary."* Pieter Wisse, http://www.wisse.cc

Metapattern of bank account



Heterachizing the OO-approach of the bank account example is giving each component some degree of contextual independence but it is not yet delivering a mechanism of mediating the parts together. Some similarities of the metapattern approach to the aspect-oriented approach are nevertheless to observe. Not only the decomposition is similar but the use of "self" understood as an empty/bare class is common. Maybe, there are no objects, but only *behaviors* in contexts/contextures.

8.3 Aspect-oriented approach to bank account

> *"AOP is not just buzzword. It's not just callback, it's not just Ruby's MixIn?, it's not just Python's metaclass, it's not just C++'s template. AOP can be implemented/considered as a subset of metaprogramming…an important subset that stands on its own. AOP deserves its name, because one really can think and program in "aspects" instead of "objects". That being said, I have never seen an example of purely aspect-based program. So, I thought I'd write up one."*
> http://mail.python.org/pipermail/python-list/2003-November/193337.html

```
class Account { }
  aspect BalanceKeeping { }
  aspect AccountStatus { }
  aspect WithdrawalLimit { }
  aspect AccountAspects : inherits BalanceKeeping,
                                  AccountStatus,
                                  WithdrawalLimit { }
 endow Account with AccountAspects
print Account.codeString()
```

"Notice that we have started with a bare class with no attributes. All the features of the class Account have been implemented by aspects, instead of interfaces or base classes."

Also this approach gives aspects independence and flexibility and the pattern is not restricting additional aspects the whole construction is subsumed under a main class, the class Account. This class is highly abstract, it has "out-sourced" all possible determinations to its aspects. Now, everything is an aspect ruled by a general object. The metapattern approach, also only descriptive and not operational, tries to implement another strategy to build identifiable objectionality: by a kind of mediation of different contexts and points of view.

Hung Jung's aspect-oriented bank account approach

```
//--------------------------------------------------
class Account {
}

//--------------------------------------------------
aspect BalanceKeeping {
    real balance
    method withdraw(amount) {
        this.balance = this.balance – amount
    }
}

//--------------------------------------------------
aspect AccountStatus {
    bool enabled
    codeblock check_status {
        if not this.enabled:
            raise AccountDisabledException
    }
    method withdraw(...) { // this meta-method is overriden later
        this.check_status
        this.withdraw.code
    }
}

//--------------------------------------------------
aspect WithdrawalLimit {
    real daily_limit
    real withdrawn_today
    codeblock check_and_update_withdraw {
        new_withdrawn_today = this.withdrawn_today + amount
        if new_withdrawn_today > this.daily_limit:
            raise WithdrawalLimitExceededException
        &inner_code
        this.withdrawn_today = new_withdrawn_today
    }
    method withdraw(...) { // this meta-method is overriden later
        check_and_update_withdraw {
            ...
            &inner_code = this.withdraw.code
            ...
        }
    }
}

//--------------------------------------------------
aspect AccountAspects: inherits BalanceKeeping,
                               AccountStatus,
                               WithdrawalLimit {
    method withdraw(...) {
        check_status
        check_and_update_withdraw { // this meta-method overrides
            ...
            &inner_code = this.withdraw.code
            ...
        }
    }
}

//--------------------------------------------------
endow Account with AccountAspects
print Account.codeString()
```

Hung Jung Lu posted this program with a well written analysis of the conception of aspect-oriented programming in contrast to OOP. It is important to see that the program starts with an abstract class without any attributes. In the AOP "ontology" attributes of objects are aspects. It should be mentioned that aspects are depending on viewpoints which can change. Thus, aspects are *behaviors* of interactional/reflectional type.

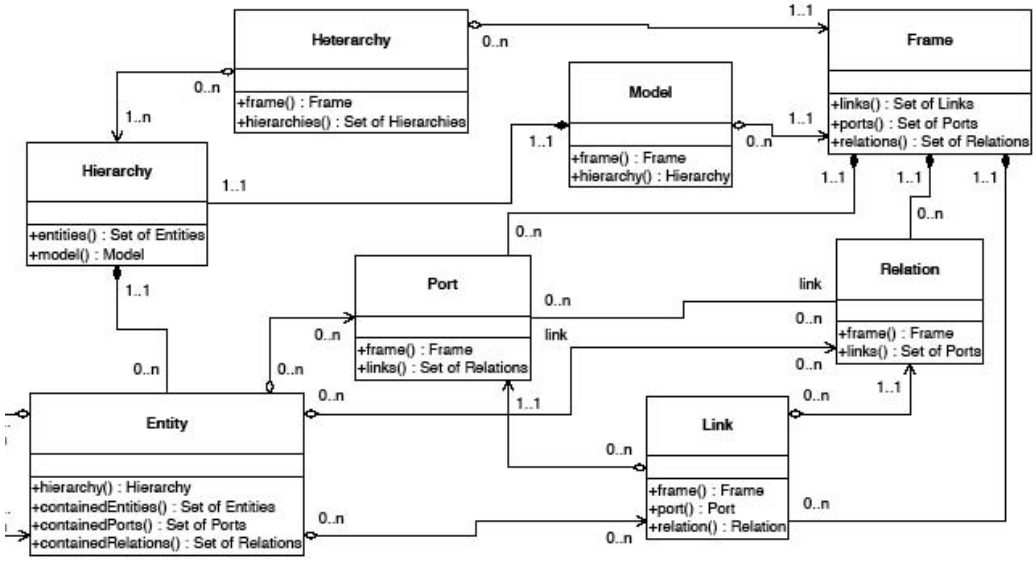
9 Towards a polycontextural approach of modeling

A key intuition underlying our work is that simple hierarchies are not rich enough to capture complex structures. In fact, we believe that any single ontological structure is insufficient. As a result, we have been exploring a variety of mechanisms that make it possible to view implement a system from multiple perspectives. One thing we believe about these multiple perspectives is that in order to have significant power they must be able to crosscut each other. That is, one perspective must be able to organize the implementation in fundamentally different ways than other perspectives.

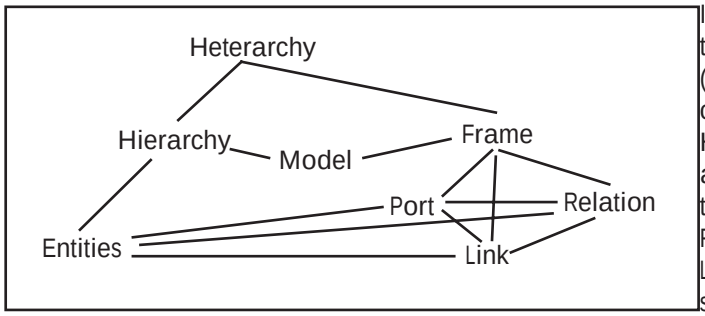
The thread that has been common to our projects is rooted in the fact that the normal rules of modularity do not suffice for describing quality software.

http://www2.parc.com/csl/groups/sda/

9.1 Heterarchy UML diagram



The conceptual graph of the UML heterarchy diagram may highlight its structure more directly.



It shouldn't be misleading to read the diagram as (methodological) hierarchy between the terms Heterarchy, Hierarchy and Entities. The additional terms Model, Frame, Port, Relation and Link are defining the structure of the interaction of the different hierarchies and their entities.

An extensive study of the history and logic of heterarchy is given by E. von Goldammer, at: http://www.vordenker.de/heterarchy/a_heterarchy-e.pdf
Limits of modularity: http://www.thinkartlab.com/pkl/media/siemens-schwarzwald.pdf

Distribution of compounds

The polycontextural approach consists of two main features. One is the *distribution* of the aspects over different contextures, the second is the *mediation* of the aspects in the polycontextural matrix. The matrix shows such a distribution, surely a highly speculative and raw one. The chosen distribution, (repl, id, id, id, id) is distributing the 5 aspects over 4 loci in the mode of identity (id) and 1 in the mode of replication (repl). For some reasons the aspect "BalanceKeeping" is in a reflectional position to the aspect "AccountStatus". The choice of the kind of distribution depends on the distinctions made in the operation of thematization. Which itself depends on the design of the architectonics, not mentioned. All together, "bank account" is designed by "thematize".

$$samba^{(5)}(repl, id, id, id, id)$$

$$\left[\begin{array}{l} chose\ style\ AOP \\ \left[\begin{array}{l} thematize\ (bank\ account) \\ \left[\begin{array}{llll} \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define\ aspect\ AccountStatus \\ \left(\begin{array}{l} lambda \\ \{statements\} \end{array}\right) \end{array}\right) \end{array}\right] & \varnothing & \varnothing & \varnothing \\ \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define\ aspect\ BalanceKeeping \\ \left(\begin{array}{l} lambda \\ \{statements\} \end{array}\right) \end{array}\right) \end{array}\right] & \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define\ aspect\ WithdrawalLimit \\ \left(\begin{array}{l} lambda \\ \{statements\} \end{array}\right) \end{array}\right) \end{array}\right] & \varnothing & \varnothing \\ \varnothing & \varnothing & \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define\ aspect\ AccountAspects \\ \left(\begin{array}{l} lambda \\ \{statements\} \end{array}\right) \end{array}\right) \end{array}\right] & \varnothing \\ \varnothing & \varnothing & \varnothing & \left[\begin{array}{l} identify\ contexture \\ \left(\begin{array}{l} define\ aspect\ printAccount \\ \left(\begin{array}{l} lambda \\ \{statements\} \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

The definition of the aspects may even be the same as in the previous modeling.

What is still missing in the scheme are the concrete rules of mediation, that is, the rules of cooperation between the contexturalized aspects. This can be introduced with the operator "elect", which is electing from the position of one contexture another contexture. The operator "elect" can have different trans-contextural definitions, all based on the appropriate as-abstractions. Because each contexture is introduced as having its own logic, arithmetics, command structure, etc. a lot of "negotiations" have to be done between the contextures to be able to realize together the job of a bank account. The aspect "printAccount" may be quite heterogeneous compared to the other aspects. It easily can transform itself along its own logic into different realizations of printing.

All that sounds totally superfluous and not necessary at all. The point is the triviality of the example. Obviously, all aspects are dealing with the same kind of arithmetics to account the bank account. A more complex example could have to deal with all sorts of incommensurability where there is no common concept available. But still cooperation has to happen as mediation in accepting the differences in play.

9.2 Hierarchical decomposition vs. heterarchic thematization

Why is a departure from hierarchic concept organization such a difficult enterprise?

OOP as a classic hierarchic system based on the is/has-abstraction is organizing its concepts "naturally" by the *inheritance* property of different objects, classes, concepts in use.

In the AOP example of BankAccount this is specially emphasized by the meta-conceptual understanding of the aspect "AccountAspects".

This "second-order aspect "aspect AccountAspects" is introduced as:

```
aspect AccountAspects: inherits BalanceKeeping,
                               AccountStatus,
                               WithdrawalLimit
```

> "Notice that we have three parts to the above code: a *class* definition, an *aspect* definition, and a final part to *endow* the aspect to the class. The last part is also known as the "aspect weaver"."

```
endow Account with AccountAspects
```

Hung Jung's programming example is strictly focussed an the aspect point of view and the OOP aspects is in the background. But nevertheless, the aspects are defining a class, i.e., *class Account*. Thus, aspects are determining objects.

> AOP is a concept, so it is not bound to a specific programming language. In fact, it can help with the shortcomings of all languages (not only OO languages) that use *single, hierarchical decomposition.*
>
> Where the tools of OOP are inheritance, encapsulation, and polymorphism, the components of AOP are join points, pointcut, advice, and introduction.
> http://www.developer.com/design/article.php/3308941

Aspect-orientation is not only a concept and neutral to programming languages, but a *design pattern*, a style or stratagem of programming. Thus defining a programming paradigm.

To cross-cut models is violating the hierarchic equivalence relation of the is-abstraction. In contrast, the as-abstraction is delivering multi-equivalence relations ruling internal hierarchies and supporting at once traversal heterarchic cross-cutting.

In another wording, abjects as the mediations of objects and aspects can be considered as exemplars of a two-dimensional is-abstraction: An abject is at once an object an aspect. Thus, an abject is identified by a double identifier. It is not given in the act of a simple identification. It has to be called twice at once. And such double-calls may be even contradictory, thus not suitable for logical modeling. In contrary, morphograms as rejects, don't listen to calls at all. They are not accessible by calls. And therefore also beyond any logical approach.

Also Ruby is characterized as a multi-paradigm programming language it is basically an object-oriented (scripting) programming language based on a 1-dimensional is-abstraction, thus not enabling genuine multiple inheritance and polysemy.

Pseudo BNF Syntax of Ruby

```
PROGRAM         : COMPSTMT

COMPSTMT        : STMT (TERM EXPR)* [TERM]

STMT            : CALL do [`|' [BLOCK_VAR] `|'] COMPSTMT end
                | undef FNAME
                | alias FNAME FNAME
                | STMT if EXPR
                | STMT while EXPR
                | STMT unless EXPR
                | STMT until EXPR
                | `BEGIN' `{' COMPSTMT `}'
                | `END' `{' COMPSTMT `}'
                | LHS `=' COMMAND [do [`|' [BLOCK_VAR] `|'] COMPSTMT end]
                | EXPR

EXPR            : MLHS `=' MRHS
                | return CALL_ARGS
                | yield CALL_ARGS
                | EXPR and EXPR
                | EXPR or EXPR
                | not EXPR
                | COMMAND
                | `!' COMMAND
                | ARG

CALL            : FUNCTION
                | COMMAND

COMMAND         : OPERATION CALL_ARGS
                | PRIMARY `.' OPERATION CALL_ARGS
                | PRIMARY `::' OPERATION CALL_ARGS
                | super CALL_ARGS

FUNCTION        : OPERATION [`(' [CALL_ARGS] `)']
                | PRIMARY `.' OPERATION `(' [CALL_ARGS] `)'
                | PRIMARY `::' OPERATION `(' [CALL_ARGS] `)'
                | PRIMARY `.' OPERATION
                | PRIMARY `::' OPERATION
                | super `(' [CALL_ARGS] `)'
                | super
```

And more!!!

http://www.ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/yacc.html

short:

```
PROGRAM         : COMPSTMT
COMPSTMT        : STMT (TERM EXPR)* [TERM]
STMT EXPR CALL COMMAND  FUNCTION
```

(Almost) Everything is a Message

All computation in Ruby happens through:
- *Binding names to objects (assignment)*
- *Primitive controls structures (e.g. if/else, while) and operators (e.g. defined?)*
- *Sending messages to objects*

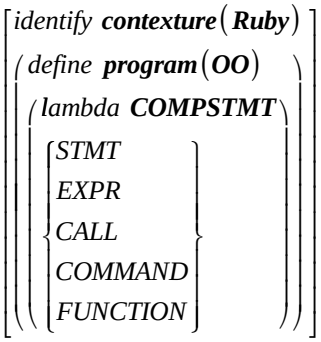
Everything in Ruby that can be bound to a variable name *is a full-fledged object.*

But there are limits, too:

- Variable names *are not objects*
- *You cannot have a variable reference another variable*
- *(no indirection to variables)*

http://onestepback.org/articles/10things

Everything is object.
Ruby is the pure object-oriented language from the beginning. Even basic data like integers are treated uniformly as objects.
ruby-man-1.4/preface.html




```
Ruby::
PROGRAM   : COMPSTMT
COMPSTMT  : STMT (TERM EXPR)* [TERM]
STMT|EXPR|CALL|COMMAND|FUNCTION
```

Ruby vs. ARS

It may be of interest to compare those limits to the approach of ARS which seems not to be restricted by such a lack of a certain self-referentiality. Ideas to polycontextural programming language, like ConTeXtures, are inspired by the approach of ARS.

ARS (Abstraction+Reference+Synthesis) provides a base for imperative programming and object-oriented programming as well and can be applied to programming in almost any programming language.
The generalization of the Lambda Calculus consists in defining the concept of abstraction simply by `give something a name'. The name hides all the details of the defined. Abstraction thus defined requires an explicit definition of a name.
The Lambda Calculus does not allow for an explicit definition of a name. The only possibility to associate a name to a value in the Lambda Calculus is by calling a function with an argument. This operation corresponds to the synthesis operation however and not to the creation of an abstraction. Lambda-abstractions in the Lambda Calculus are `per se' anonymous.
http://www.aplusplus.net/bookonl/node18.html

ARS vs. ConTeXtures

ARS is based and founded on uniqueness. ConTeXtures are involved in multitudes. Additional to ARS's 3 principles of Abstraction, Reference and Synthesis, ConTeXtures are introducing a fourth principle of *dissemination* as an interlocking mechanism of *distribution* and *mediation*, thus: $\text{ConTeXtures}^{(m,\,n)} = \text{diss}^{(m,\,n)}(\text{ARS})$.

10 Diamond Strategies of Programming

"Abject" is always negative, meaning "lowly" or "hopeless." You can't experience "abject joy" unless you're being deliberately paradoxical.
http://www.wsu.edu/~brians/errors/abject.html

10.1 Preliminary steps

Inheritance vs. mediation

Mediation as a heterarchic thematization is ruled by the proemial relation. It is not simply an additional hierarchy, say a queer, horizontal, traversal, orthogonal hierarchy resulting in a double hierarchy of objects and aspects, realized in OOPxAOP. The heterarchic interpretation of aspects, cross-cutting abstraction, as I try to thematize is surely a radicalization of the existing activities and not a genuine interpretation of AOP. Thus, e.g, it will not confirm the statement: *"The form of AOP presented here is much more powerful than the original proposal, fits in well with Ruby's overall design and use, and so will nicely compliment the Ruby Way."*

It is not enough to introduce additionally to objects the concept of aspects. If the introduction of aspects is not reflected by a new concept which is mirroring the interplay between objects and aspects there is no other chance than to linearize or hierarchize the story under one of the categories. Mostly, the category object will dominate the category aspects. In other word, the whole story is brought back home under the roof of the general class as the common root of object and aspect. Such a homogenization is unavoidable because of the underlying mono-contexturality of the whole programming paradigm.

http://www.thinkartlab.com/pkl/media/callen.htm

Hierarchy and heterarchy

The chiasm under consideration is the proemiality between aspects and objects.

Heterarchic strategies are modeled by *aspects*, hierarchic by *objects*. Both together in their togetherness are realized as *abjects*. Both in their processuality as mutual rejections of structurality of polycontexturality are inscribed as morphograms, short: *rejects*. Rejects are inscribing the processuality of the actions of objects and aspects thematized from their polycontexturality. Morphograms are inscribing processuality in abstracting/subversing any contexturality. Thus morpgograms/rejects are focussing on two different aspects/features of processuality. The processuality of objects is realized as classification/categorization/generalization/encapsulation. The processuality of aspects is realized as traversing/cross-cutting/weaving. Thus, AOP should be considered properly not mainly as an aspect-centered approach but as a mediation of aspects and objects, thus OOPxAOP.

Efficiency of AOP, concerning programming code, seems to be very impressing, reducing OOP code decisively. This shouldn't come as a surprise. To work at once in 2-dimensions, programming vertically and horizontally together, is of a much stronger abstraction than programming in a 1-dimensional approach. If designed badly it can rapidly change into its opposite and producing confusion.

Diamond Programming patterns

Aspects of aspects, aspects as aspects, or Aspectivity of aspects,

Aspects of objects, aspects as objects, or Aspectivity of objects,

Objects of aspects, objects as aspects, or Objectionality of objects,

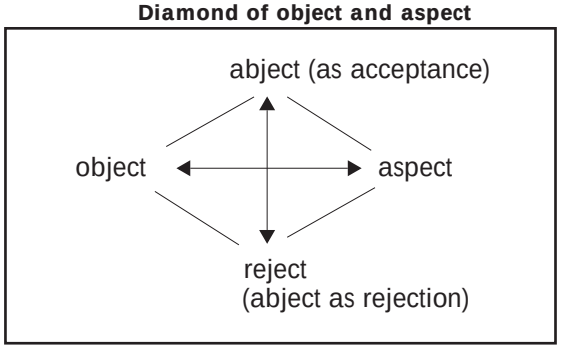
Objects of objects, objects as objects, or Objectionality of aspects.

Abjects as both at once: objects and aspects,

Rejects as neither objects nor aspects, but still in the game of both.

It may be reasonable to focus mainly on objects but it may as well be reasonable to focus manly on aspects. Nevertheless, both are two different dimension of a mediated system. A conscious programming should be aware and take into account all features of the diamond programming pattern. From the point of view of object-oriented programming aspects are special objects. From the point of view of aspect-centred programming objects are special aspects. But this is an unnecessary reduction.

Diamond of object and aspect



With the invention of polycontexturality the interplay between objects and aspects can be modeled without denying the autonomy of both categories. Abjects as mirrors of this interplay are not a new super-category or super-class but a mediating part of the game. Abjects are neither objects nor aspects. As mirrors they are at the same time both at once, objects as well as aspects. The difference of the richness of the *acceptance* of both, *abjects*, and the poverty of *rejection* of both producing *rejects* is not yet thematized at this step of conceptualization. That is, in the following, I will study the interacting diamond of (object, aspect, abject) only. I also exclude iterations and accretions of the diamond pattern. However, it has to be seen that such an interdependency is purely functional. Aspects and abjects can be objectized by objects, abjects and objects can be aspectized by aspects. Also, objects and aspects are abjectized (mirrored) by abjects.

Abjects are over-determined, antagonistic, polyvalent terms. Rejects are, in some sense, under-determined, or even zero-determined terms. Rejection, in its radicality is accessible by the morphogrammatic abstraction which is eliminating any kind of semantics. In a less radical sense, rejects are acting between contextures. They are rejecting a dual situation, like object/aspect, for a different category, say inject or project which belongs to another contexture.

In other words, additional to the "substantial" conceptual strategy of object- and aspect-thematization which are usually in the fore-ground of programming, the functional interplay of all three categories is of importance and can change from its back-ground to its fore-ground function. Such conceptual and computational dynamics are "minimal conditions" to a new design of programming paradigms.

Tectonics of AOP

Methods: computational components,

Objects: classificational components,

Aspects: cross-cutting components.

With this simplified tectonics of AOP a complex interlocking mechanism can be studied. Between all categories, the methods, objects and aspects, the combinatorics of proemiality is playing the pragmatics of programming, design and development.

polycontextural/morphogrammatic programming framework

Why should we develop a polycontextural/morphogrammatic programming paradigm? Once an approach is well working, like OOP or as a special example, Ruby, new demands, new perspectives of programming occurs. Then, the paradigm has to be changed, fixed, modernized to the scope with the new demands. Obviously, there are two strategies to chose. One is to change the practical focus of programming while still accepting the existing paradigm. The other is to change, adapt the language to the new scenario. But this may turn out to be not very easy and probably also not specially efficient and helpful.

> *"So AOP is very important to me and I have been looking forward to having this as a core feature of Ruby." From what I have read AOP should not be a core feature of Ruby because it is too complex. Ruby should be capable of doing AOP but I would not change anything fundamental about the Ruby creation process to make this a core feature. -- Bob*
>
> Relpy: *Understandable, and I assure you we are working to minimize any such changes. But AOP cannot be properly done without some changes to core Ruby, as has been recently demonstrated.*
> http://www.rubygarden.org/ruby?AspectOrientedRuby/NotesRoughDrafts

And what happens next? There will be again a new demand on the way waiting to be implemented and realized in a newly renewed programming language.

Polycontexturality/morphogrammatics could offer a different approach. Because polycontexturality with its heterarchic organization new demands could be added without any conceptual problems as new contextures. New contextures have to be mediated to the previous ones, but there is no need for a full change of all and everything. This is not denying intrinsic difficulties and also not the necessary super-additivity of any extension. Extensions can be seen as fusion, merging and evolutions.

The constellation I introduced in this paper is focussed on reflectionality/interactionality of computation which doesn't mean a systematic restriction to a 2-dimensional approach at all. Additional categories of behaviors would be covered by *intervention* and *interlocution*. For other purposes, as many other "dimensions" as necessary can be introduced because polycontexturality is itself introduced by a multitude of mediated contextures. Also, to speak about dimensions and dimensionality can be misleading and has to be deconstructed to a more neutral topological terminology. Thus, new demands, say, of context-, connex-, subject-, environment-, view-point-, graphematics-oriented approaches could be added by mediation without destroying existing paradigms and routines. Hence, proemiality and polycontexturality is open to complex programming paradigms involving, say, objects, aspects, features, domains, generators, and so on. The book *"Generative Programming"* from Czarnecki and Eisenecker presents a very broad an deep analysis of existing programming paradigms.

Modularization of programming paradigms

This is leading to a modularization of programming paradigms (styles). A modern text is living of its intertextuality of different styles. Intertextuality of texts are the necessary condition for meta-reflection and self-modifications. Modularization from objects to aspects and more, and many to come, to modularization of programming and designing paradigms. Classic modularization principles are still governed by the metaphor and reality of classic computation. The model of classic computation is the Turing Machine. Turing Machine computation is bare of any interactionality/reflectionality in the strict sense (Peter Wegner). As much as aspects has to be woven and compiled together, the relative autonomous paradigm have to be mediated and brought together by a new kind of compiler system.

Morphogrammatics

Morphogrammatics is just defined as rejection of existing concepts or stratagems, like objects and aspects. Thus, it is not restricted in any way by the paradigmatic definitions of multi-programming paradigms. The subversion of morphogrammatics is still difficult to think and to apply to programming, but it may open up a "space" of structurations and transformations/metamorphosis which is independent of onto-semantic constructs, concepts and classes, and their limitations.

Thematization and reflectionality/meta-programming

Thematization as designed in polycontextural programming is highly reflectional. Classic programming is based on all sorts of abstractions. Abstractions are ontologic-concept oriented, i.e., concepts are classes, classes have attributes of real things. This kind of programming deals with real things, like BankAccount. But real problems/conflicts/antagonism today are not simple defined by things. They are complexions of intertwined refections. And fucussation of things/classes/concepts/objects are still thought in an ontological framework excluding reflectionality.

Reflection in the framework of is-abstraction is: reflection of reflection of something. Focussed finally on the "something" and not on the processuality of reflection. Seymor Pappert (Minsky, Society of Mind): "*We can not think about thinking, without thinking about something."* And the other way round: *"The relation between a picture and the thing pictured cannot be pictured."* (Wittgenstein) Both may sound quite reasonable, but they aren't. Today they are not even practical anymore. And worse, you cannot prove them at all. They are exhausted belief statements. If you make the option for *thematization* instead of abstraction, the ontology reduces to the formula: X as X is X ==> X is X. And again, there is no way back from is-abstraction to as-thematization. Is-abstraction is a reduction of thematization, and with it we are living in a reduced world.

Ontology vs. formal languages

Since the linguistic turn it is common to belief that formal languages, like logical systems, are neutral to ontology and therefore need an ontological interpretation, done mainly by semantics. Again, there is nothing wrong with that, as long as it accepts its own limits. From a transcendental-logical point of view this turn is forgetting its own prerequisites, which are its signs, collected in a sign repertoire, called alphabet. Those signs, which are the base of the ontology-neutral formalism, are themselves topics of ontological reflections, namely as sign-events or sign-entities existing in this world as sign-identities, whose laws are just the laws of (general formal) ontology (Husserl).

This stuff is well developed in: Ernst Tugendhat, Self-Consciousness and Self-Determination, MIT Press 1986 (German 1979). Another approach is: Brian Cantwell Smith, On the Origin of Objects. A. The ontology of computation, MIT 1996, http://www.formalontology.it/smithbc.htm

10.2 Pattern of paradigms for the complexion object-aspect-abject

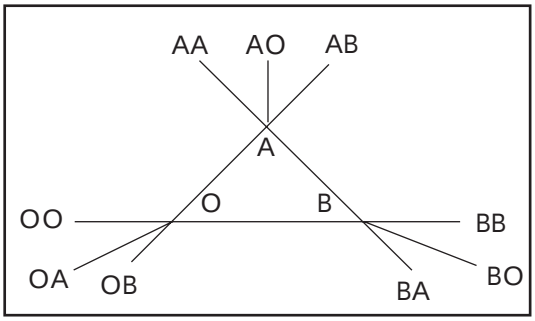
AA: Aspects as aspects, or Aspectivity of aspects,
AO: Aspects as objects, or Aspectivity of objects,
AB: Aspects as abjects, or Aspectivity of abjects
OA: Objects as aspects, or Objectionality of aspects,
OO: Objects as objects, or Objectionality of objects,
OB: Objects as abjects, or Objectionality of abjects,
BO: Abjects as objects, or Abjectivity of objects,
BA: Abjects as aspects, or Abjectivity of aspects,
BB: Abjects as abjects, or Abjectivity of abjects.

	object	*aspect*	*abjects*
object	*OO*	*OA*	*OB*
aspect	*AO*	*AA*	*AB*
abjects	*BO*	*BA*	*BB*

$$\underbrace{object - aspect}_{first-level} \qquad \underbrace{abject}_{second-level}$$

An object in general, object(3), thus is a complexion of the components object, aspect, abject. The same for aspects(3), and abjects(3).

The new topic under computational consideration is a complexion mediating the categories object, aspect and abject together. Semiotically it is a triadic-trichotomic sign-complexion in the sense of Charles Sanders Peirce.



Objects(3):
O-objects, A-objects, B-objects
Aspects(3):
O-aspects, A-aspects, B-aspects,
Abjects(3):
O-abjects, A-abjects, B-abjects.

Collecting the components.

Historical remark: After more than 20 years

In the research report OVVS we have given 1985 a very first theory and model of the production process at a computer manufacturer (Siemens AG, Augsburg, Germany) in a polycontextural setting. We thematized the process of production of mainframes from a polycontextural and semiotic point of view. At this time OOP was just arriving and everything had to be put into its terminology. But our analysis has shown very clearly the serious limits of the OOP approach in programming and as an epistemological model. Not worth to mention, that such an endeavour was totally displaced in time and ideology. Therefore, we didn't got the chance to elaborate our seminal approach more properly. Today (2006) the situation sounds strikingly familiar. The missed chances, but also the lack of further explanations of our study springs badly into the eyes, too.
Without being aware about the repetition of history I'm writing again some post-objectional constructions. At least, the limits of OOP are obvious now for some people. And ideas of plurality, multitudes, interaction, reflection and mediation are growing rapidly. In some sense.

OVVS, Munich 1985:
http://www.thinkartlab.com/pkl/media/siemens-schwarzwald.pdf

10.3 Augmenting complexity of Thematizations

Complexity of thematizations, or dimensionality of distribution, or topology of disseminations, can naturally be augmented by adding new contextures. It shouldn't be forgotten that such an augmentation is super-additive, thus 3 elementary contextures are producing 3 additional mediating contextures on higher level.

$$\underbrace{object - aspect - inject}_{first-level}$$

$$\underbrace{abject}_{second-level} \quad \underbrace{perspect}_{second-level}$$

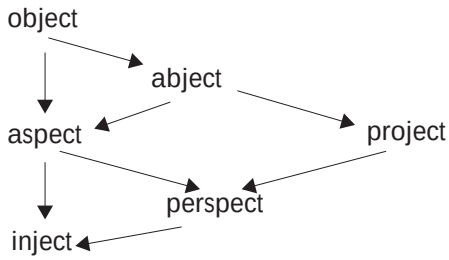
$$\underbrace{project}_{third-level}$$

objects, aspects and injects (views) are on the same first level of thematization, abjects and *perspects* are reflecting the thematizations of the first level, and on a third level projects are reflecting the thematizations of the second level, thus projects are reflecting the activity of the whole system.

The "shading" (Abschattung, Husserl) or *aspectication* of all kinds of thematizations together may be collected, at first, in the following matrix. In this matrix a first-order shading only is realized. That is, possibilities of higher-order constellations like OAB, AOPB, etc. are not mentioned.

	object	*aspect*	*abject*	*inject*	*perspect*	*project*
object	*OO*	*OA*	*OB*	*OI*	*OP*	*OJ*
aspect	*AO*	*AA*	*AB*	*AI*	*AP*	*AJ*
abject	*BO*	*BA*	*BB*	*BI*	*BP*	*BJ*
inject	*IO*	*IA*	*IB*	*II*	*IP*	*IJ*
perspect	*PO*	*PA*	*PB*	*PI*	*PP*	*PJ*
project	*JO*	*JA*	*JB*	*JI*	*JP*	*JJ*

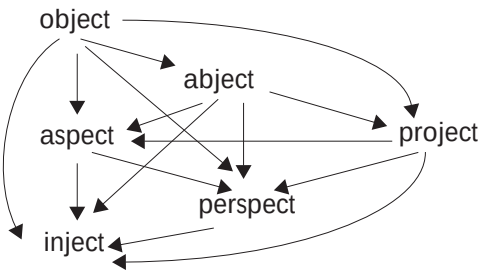
10.3.1 Conceptual graphs of project



This is not the place to go into the theoretical problems of defining a complex theory of signs which has to surpass the triadicity of semiotics. Triadic-trichotomic structures are well modeled by 1-categories. Mediations of such structures are better modeled in n-categories.

The graph shows the commutative relations between different terms.

A complex entity, a complexion, is composed of its constituents as given by the conceptual graph. Also the graph may not show it clear enough, there is no fixed root. All components are of equal relevance. And each can play to be a root–or not. Thus, object is not the origin, it may be a beginning. But there are others, too.

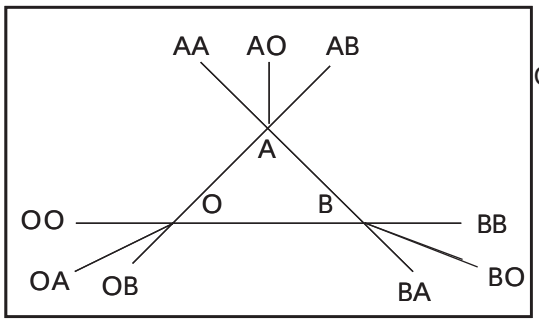


10.3.2 Composition/decomposition and duality of conceptual graphs

The complete graph of *project* can be decomposed into its triadic-trichotomic subgraphs. On the other hand, given the decomposites, the whole can be composed.

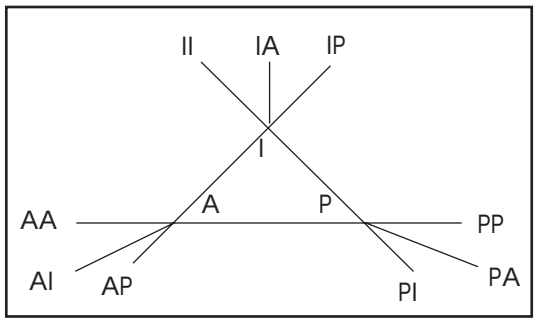
Between (X as Y) and (Y as X) a reflectional *duality* exists. Thus, objects can be seen as (O-objects, O-aspects, O-abjects) or dual as (O-objects, A-objects, B-objects).

Object-aspect-abject-diagram



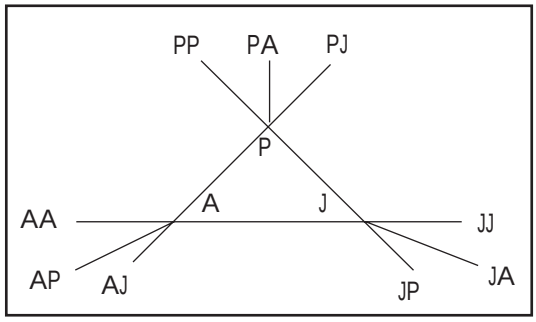
Objects(3):
O-objects, O-aspects, O-abjects
Aspects(3):
A-aspects, A-objects, A-abjects,
Abjects(3):
B-abjects, B-objects, B-aspects.

Aspect-inject-perspect-diagram



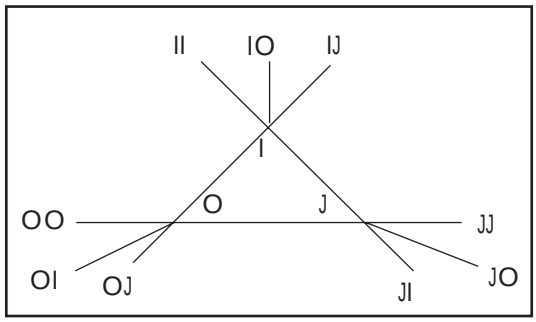
Aspects(3):
A-aspects, A-injects, A-perspects
Injects(3):
I-injects, I-aspects, I-perspects,
Perspects(3):
P-perspects, P-injects, P-perspects.

Abject-perspect-project-diagram



Abjects(3):
A-abjects, A-perspects, A-projects
Perspects(3):
P-perspects, P-aspects, P-projects,
Projects(3):
J-projects, J-abjects, J-perspects.

Object-inject-project-diagram



Objects(3):
O-objects, O-injects, O-projects
Injects(3):
I-injects, I-objects, I-projects,
Projects(3):
J-project, J-objects, J-injects.

10.3.3 Duality principle in triadic-trichotomic diagrams

Object-aspect-abject-diagram

	object	*aspect*	*abjects*
object	*OO*	*OA*	*OB*
aspect	*AO*	*AA*	*AB*
abjects	*BO*	*BA*	*BB*

This 3x3-matrix of the categories object, aspect and abject gives the 9 combinatorial possibilities of their mutual thematizations in the mode of the as-abstraction. Object as object, object as aspect, etc. All together necessary to describe the AOP-system semiotically as a whole.

Self – dual constellations	*dual constellations*
$dual(OO) = OO$	$dual(AO) = OA$
$dual(AA) = AA$	$dual(BO) = OB$
$dual(BB) = BB$	$dual(BA) = AB$

Objects(3): O-objects, O-aspects, O-abjects: (OO, OA, OB)
Aspects(3): A-aspects, A-objects, A-abjects: (AA, AO, AB)
Abjects(3): B-abjects, B-objects, B-aspects: (BB, BO, BA)

dual(OO, OA, OB) = (OO, AO, BO), thus object(3)
dual(AA, AO, AB) = (AA, OA, BA), and aspect(3) and
dual(BB, BO, BA) = (BB, OB, AB), and abject(3) have a dual interpretations.

Aspect as object. dual. object as aspect
Abject as object. dual. object as abject
Abject as aspect. dual. aspect as abject.

10.3.4 Self-dual constellations

There are a lot of interesting features given by the semiotic matrices and their operators. One more is the self-duality of constellations.

dual (OO, AA, BB) = (OO, AA, BB)
dual(BO, AA, OB) = (OB, AA, OB)

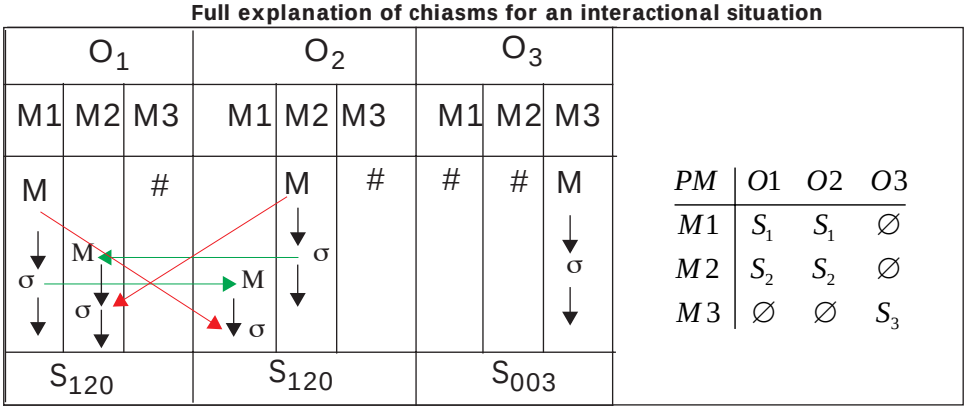
This short semiotic reflection, exemplified at the object-aspect-abject-diagram, shows clearly that our entities (object, aspect, abject) are not in any sense classes with some attributes. And questions about adequacy of attributes making them true or false are not in the game at all. As-abstractions and its operations, like dualization, are second-order reflectional concepts. Thus, their underlying logic, polycontextural logic, is not a logic of truth and falsehood but a logic of reflection (and interaction).

Semantic or ontological questions are not lost but they appear on a first-order level where we deal with the concepts in an isolated way, say as: object is object, aspect is aspect and abject is abject, using the is-abstraction, and they will have some identifiable attributes or not.

Well, if programming is conceptual modeling, we made an important step forward to a modeling of reflectional / interactional, cognitive and volitive, situations not accessible until now to first-order concepts and techniques of modeling and programming. *"Cognition and Volition"*: http: / / www.vordenker.de / ggphilosophy / c_and_v.pdf

10.4 Proemiality between aspects and objects

Full explanation of chiasms for an interactional situation



PM	O1	O2	O3
M1	S_1	S_1	∅
M2	S_2	S_2	∅
M3	∅	∅	S_3

The wording here is not only *"types becomes terms and terms becomes types"* but *"a type as a term becomes a term"* and, at the same time, "*a type as type remains a type"*. Thus, *"a type as a term becomes a term and as a type it remains a type".* And the same round for terms. This modeling is easy applied to an aspect/object-chiasm.

Full wording for a chiasm between terms and types over two loci

Explicitly, first the green round,

"A type $\sigma^{1.1}$ *as a term* $M^{2.1}$ *becomes a term* $M^{2.1}$ *and as a type* $\sigma^{1.1}$

it remains a type $\sigma^{1.1}$*".*

And,

"A type $\sigma^{2.2}$ *as a term* $M^{1.2}$ *becomes a term* $M^{1.2}$ *and as a type* $\sigma^{2.2}$

it remains a type $\sigma^{2.2}$*".*

And simultaneously, the second round in red, the same for terms:

"A term $M^{1.1}$ *as a type* $\sigma^{2.1}$ *becomes a type* $\sigma^{2.1}$ *and as a term* $M^{1.1}$

it remains a term $M^{1.1}$*".*

And,

"A term $M^{2.2}$ *as a type* $\sigma^{1.2}$ *becomes a type* $\sigma^{1.2}$ *and as a term* $M^{2.2}$

it remains a term $M^{2.2}$*".*

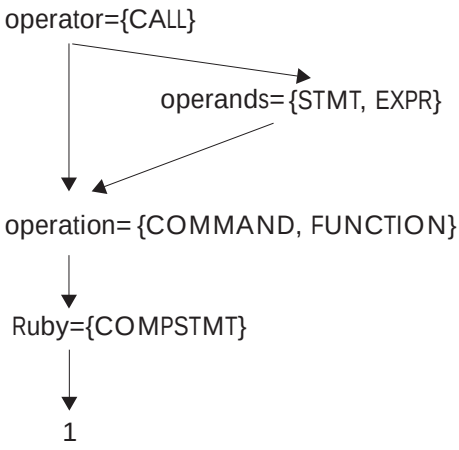
And finally, between terms $M^{1.1}$ and $M^{2.2}$, and types $\sigma^{1.1}$ and $\sigma^{2.2}$, a categorial coincidence is realized. To round up, the same coincidence holds for terms and types of $LC^{1.2}$ and $LC^{2.1}$. Thus, a type has two functions at once, a type as a type and a type as a term. Therefore, this double meaning has to be distributed over different localization of the complex constellation. Otherwise it simply would produce unnecessary conflictive overlapping. The *matrix* shows clearly the kind of distribution, the *diagram* is visualizing the process of the chiasm involved. A more formal treatment can be found at: http://www.thinkartlab.com/pkl/lola/poly-Lambda_Calculus.pdf

11 Rudy; some Dissemination of Ruby

11.1 Conceptual graph of Ruby

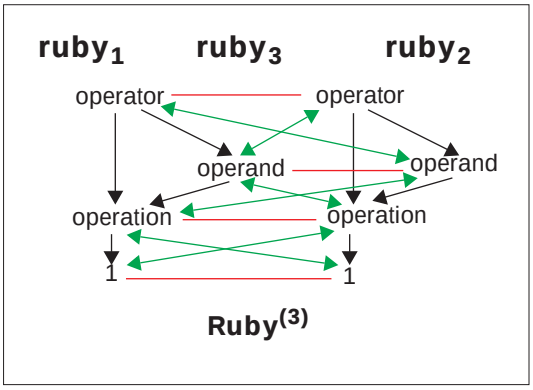
$$\left[\begin{array}{l} identify\ \textbf{\textit{contexture}}(\textbf{\textit{Ruby}}) \\ \left(\begin{array}{l} define\ \textbf{\textit{program}}(\textbf{\textit{OO}}) \\ \left(\begin{array}{l} lambda\ \textbf{\textit{COMPSTMT}} \\ \left\{\begin{array}{l} STMT \\ EXPR \\ CALL \\ COMMAND \\ FUNCTION \end{array}\right\} \end{array}\right) \end{array}\right) \end{array}\right]$$

```
Ruby::
PROGRAM   : COMPSTMT
COMPSTMT  : STMT (TERM EXPR)* [TERM]
STMT|EXPR|CALL|COMMAND|FUNCTION
```



An operator terminology may summarize Ruby as COMPSTM with {STMT, EXPR} as operands, {CALL} as operators and {COMMAND, FUNCTION} as operations. Ruby then is based on its uniqueness, represented by 1.

Distribution of 3 ruby to one big Ruby$^{(3)}$.



ruby3 is not written explicitly.

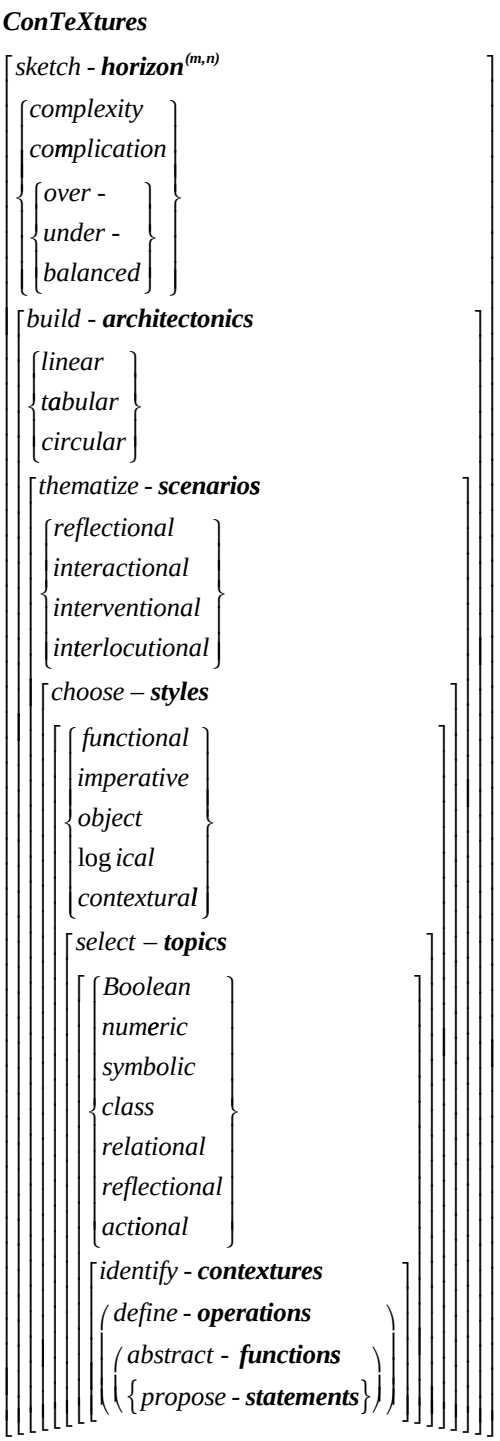
$Ruby^{(m)}$	O_i	O_{i+1}	O_{i+2}
M_i	$ruby_1$	$ruby_2$	$\varnothing$
M_{i+1}	$ruby_1$	$ruby_2$	$\varnothing$
M_{i+2}	$\varnothing$	$\varnothing$	$ruby_3$

General distribution pattern of reflectional $ruby_1$ and reflectional $ruby_2$ and computational $ruby_3$.

Diamonds of Rubies.

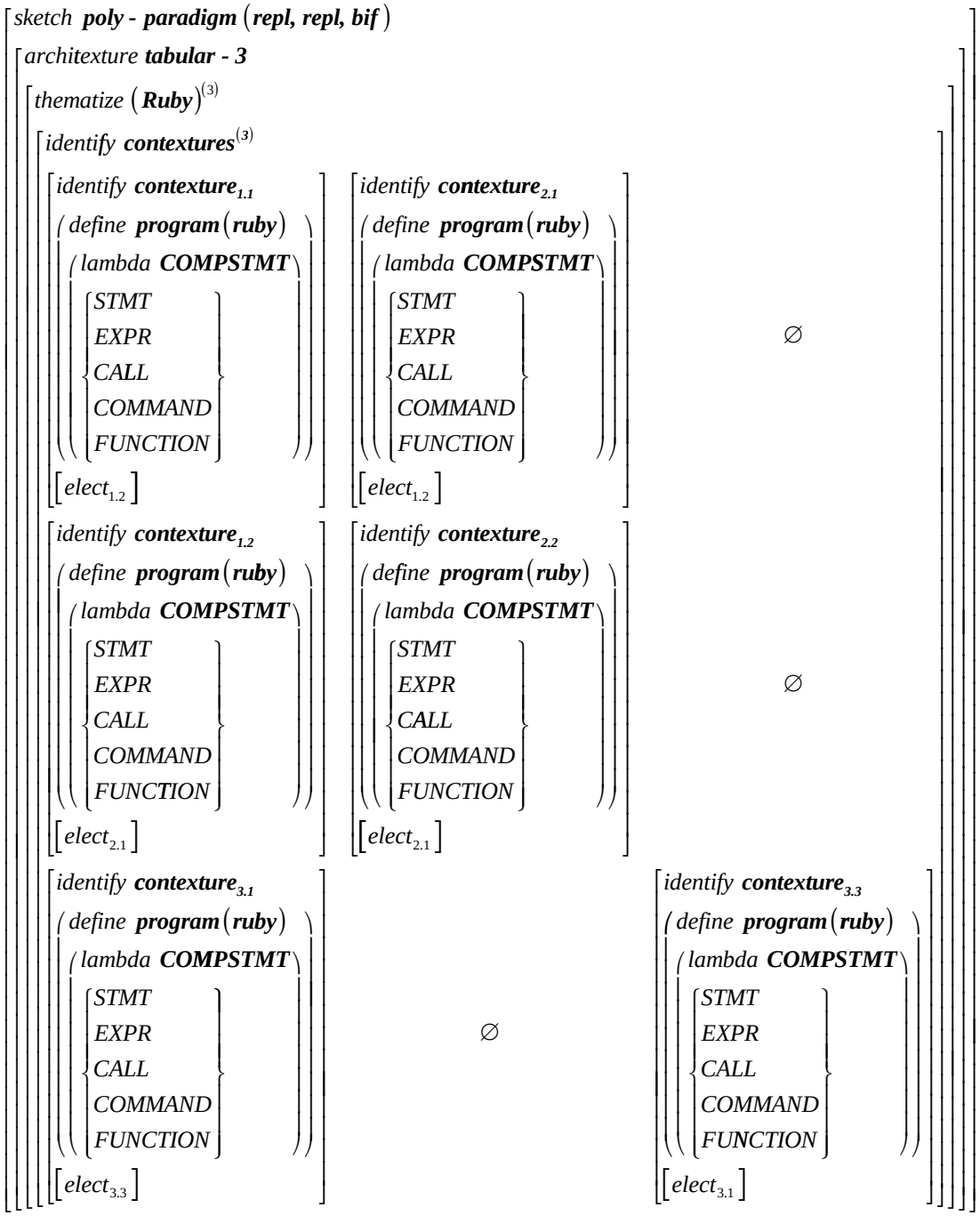
11.1.1 General poly-paradigm scheme Rudy as introduced for ConTeXtures

This general scheme handles a heterarchic distribution of different paradigms of programming with their different scenarios, styles and topics based on complex architectonics as sketched by the general horizon of thematization and computation.



11.2 Example of a 3-contextural dissemination of Ruby

Reflectional and interactional distribution of the Ruby programming language over different contextures. Additional to the dissemination of Ruby the operators *elect* are ruling possible chiasms between intra- and trans-contextural programming constructs.



11.3 Global and local structure of disseminated Ruby

At all occurrences in the distribution matrix of the disseminated Ruby programming language all of its constituents are repeated and realized. All constituents and all laws are placed intra-contexturally at their loci which, in this case, are designed by the reflectional and interactional dimension of the architectonics. This can be called the *intra-contextural* inheritance of the distributed systems and it represents the *local* thematization of the complex programming system. Thus, this distribution is, at first, homogenous. Only the same kind of languages are distributed, i.e., the language of Ruby, thus Ruby and no mix with other languages.

General scheme for identical mappings

$$Ruby^{(m)} : \left[Ruby^{(m)} \right]_{refl,\ act} \xrightarrow[sops]{} \left[Ruby^{(m)} \right]_{refl,\ act}$$
$$id : \forall i, j \in s(m): \qquad \left(Ruby^{i,j} \right) \xrightarrow[id]{} \left(Ruby^{i,j} \right)$$

As a complexion Ruby(3) has to be thematized from its *global* behavior. This is the place where the so called *super-operators* (sops) enter the game. These super-operators are ruling the game between different contextures in the modi of identity, replication, permutation, reduction and bifurcation.

Super-operators for distributed Ruby

$$Ruby^{(m)} : \left[Ruby^{(m)} \right]_{refl,\ act} \xrightarrow[sops]{} \left[Ruby^{(m)} \right]_{refl,\ act}$$
$$sops = \{id,\ perm,\ red,\ bif,\ repl\}$$
$$id : \forall i, j \in s(m): \qquad \left(Ruby^{i,j} \right) \xrightarrow[id]{} \left(Ruby^{i,j} \right)$$
$$perm(i,\ j): \forall i, j \in s(m): \left(Ruby^{i}\ ,\ Ruby^{j} \right) \xrightarrow[perm]{} \left(Ruby^{j},\ Ruby^{i} \right)$$
$$red(i,\ j): \forall i, j \in s(m): \left(Ruby^{i},\ Ruby^{j} \right) \xrightarrow[red]{} \left(Ruby^{i},\ Ruby^{i} \right)$$
$$bif(i,\ j): \forall i, j \in s(m): \left(Ruby^{i},\ Ruby^{j} \right) \xrightarrow[bif]{} \left(\left(Ruby^{i} \parallel Ruby^{j} \right),\ Ruby^{j} \right)$$
$$repl(i,\ j): \forall i, j \in s(m): \left(Ruby^{i},\ Ruby^{j} \right) \xrightarrow[repl]{} \left(\left(Ruby^{i} \mid Ruby^{i} \right),\ Ruby^{j} \right)$$

It turns out that the special case of an identical mapping is part of the general super-operators. Even for the case of a single occurrence of Ruby, that is, for m=1. This positioning of a system or language, even for only once, as a mapping onto itself is crucial and can be omitted only because of its singularity. Not to be aware of it is called the *Blind Spot* of the system.

More discussion of Ruby

Ruby the Rival by Chris Damson, 11/16/2005
http://www.onjava.com/pub/a/onjava/2005/11/16/ruby-the-rival.html?page=1
James Duncan Davidson: Trying Something New
"And I think that's the real win from the recent attention on Ruby on Rails and the breakaway from viewing the world with Java-colored glasses. It's not that Ruby on Rails is going to be the next Java. Far from it. It's that Ruby on Rails helps to break this idea that there is "One True Way." There's not. There are many different ways to solve a problem. And really, none of them is the clear-cut winner. There's just places where one solution has advantages."

Also, Ruby, Postmodernism, Java, Lambda:
http://lambda-the-ultimate.org/node/1123

Including the tectonics of the Ruby language into the polycontextural mapping we get the general scheme as below.

$$
\begin{aligned}
&Ruby^{(m)} : \left[Ruby^{(m)}\right]_{refl,\ act} \xrightarrow[sops]{} \left[Ruby^{(m)}\right]_{refl,\ act} : \\
&STMT^{(3)} : STMT^{(3)} \xrightarrow[sops]{} STMT^{(3)} \\
&EXPR^{(3)} : EXPR^{(3)} \xrightarrow[sops]{} EXPR^{(3)} \\
&CALL^{(3)} : CALL^{(3)} \xrightarrow[sops]{} CALL^{(3)} \\
&COMMAND^{(3)} : COMMAND^{(3)} \xrightarrow[sops]{} COMMAND^{(3} \\
&FUNCTION^{(3)} : FUNCTION^{(3)} \xrightarrow[sops]{} FUNCTION^{(3} \\
&sops = \left\{id,\ perm,\ red,\ bif,\ repl\right\}
\end{aligned}
$$

This modeling is conservative in many senses, accepting the mapping from constituents to constituents as type trusty. There is no mapping from one type into another type considered. Such "untrustful" mappings are part of *metamorphic* transformations in complex formal systems.

11.3.1 Data structure of Ruby$^{(3)}$

Some information about the involved data structure as objects, aspects and abjects was introduced before. This has to be concretized in respect of its syntax with statements and expressions.

STATEMENTs and EXPRESSIONs

Polycontextural systems are not based on statements and expressions but on intertextuality between textual systems containing statements and expressions.

It is supposed that STATEMENT is mapped onto STATEMENT and in the same way EXPRESSION onto EXPRESSION. Thus, on this level of conceptualization no chiastic crossings between different categories, like STATEMENT and EXPRESSION, are involved. But again, those mappings are ruled by the super-operators *sops*.

$$
\begin{aligned}
&Ruby^{(m)} : \left[Ruby^{(m)}\right]_{refl,\ act} \xrightarrow[sops]{} \left[Ruby^{(m)}\right]_{refl,\ act} \\
&STMT^{(3)} : STMT^{(3)} \xrightarrow[sops]{} STMT^{(3)} \\
&EXPR^{(3)} : EXPR^{(3)} \xrightarrow[sopsb]{} EXPR^{(3)}
\end{aligned}
$$

11.3.2 Call structure

Calls for COMMANDs and FUNCTIONs have to be distinguished as intra-contextural calls and trans-contextural calls which are obviously given with the super-operators of bifurcations and the architectonics of reflectionality and interactionality. Intra-contextural topics are not considered in this proposal because they correspond more or less to the well known topics of the language under consideration, here Ruby.

11.3.3 Command structure of Ruby(3)

The command structure was until now only given by reflectional and interactional distributions and electors. Both are realized by the so called super-operators

sops ={id, repl, perm, red, bif}.

Further information is directly produced by the underlying logical system of the disseminated programming language. This polycontextural logic is formalized as PolyLogics. Additional to the distributed junctions and deduction rules, a complex apparatus of transjunctional and multi-negational operators are included.

Thus, COMMAND is distributed as COMMAND(3) with junctions, negations, deduction rules (IF-THEN) and a family of transjunctions distributed over contextures.

11.3.3.1 If-Then-Else control structure disseminated

Lessons from ConTeXtures about *electors* and *selectors*.

If-abstraction in general:

One of the most basic operation in any programming language is to make a decision, to select a block of code depending on the truth value of a certain argument.
Such an operation people have in mind, when they talk about IF-statements, IF-THEN-ELSE-constructs, alternative structures and a few more.
We can very well say that the IF-statement has the function to select code to be evaluated or executed. The IF-statement therefore is a function taking three arguments:

1.a condition having a certain truth value (true or false),
2.the first block of code and
3.the second block of code.

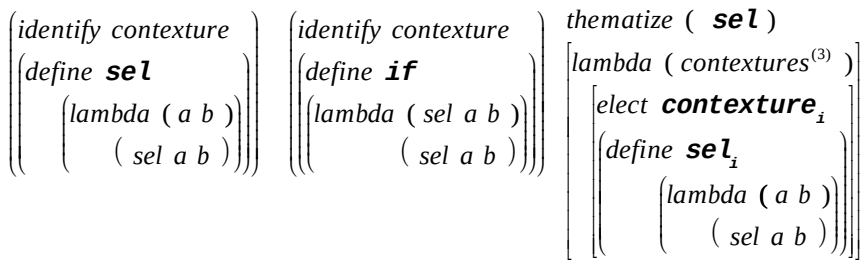
Because the selection of the code block to be evaluated or executed depends on the first argument, we can look at the first argument as a selector.

Having thus analyzed the essence of an IF-function we may code it as a `lambda abstraction':

```
(define if ( lambda ( sel a  b)
                    ( sel a b )))
```

Remark: eager/lazy
Many programming languages have problems with such a construct, because their language implementation evaluates the arguments before they are effectively passed to a function. This is in contrast to the function of the selector to select the code to be evaluated.
http://www.aplusplus.net/bookonl/node74.html

Defining *sel* **Defining *if*** **Defining *sel* with *elect***



This is first the classic definition/introduction of the *selector* operator which is basic to define the if-construct but it is set into a contexture which itself has to be identified.

Otherwise, as we know, all falls from the sky into our hands. There is, additionally, as usual, some circularity involved. Thus, IF is defined by the selector operation which is based itself on the operator *sel*.

To identify a contexture also needs a selecting operation. It may be called *elector* (elect). Thus, given or chosen the elector we can define the selector more properly.

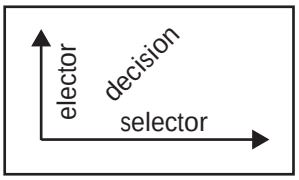
Because of the graphematic tabularity of ConTextures there are not *"only two possibilities to perform a selection"*. For each contexture there are intra-contexturally only two possibilities to perform a selection. But between contextures, trans-contexturally, there are as many new selectors as neighbor contextures. These new "selectors" should be called *electors.* Electors are electing the election for selectors to perform each mutually its selection.

That is, a selection can happen at once at different loci of a disseminated systems. In other words, such a general selector as any other successive or procedural action has to be realized in the two dimensions of intra- and trans-contexturality. Thus, a selector as a single operation can be realized intra-contextural staying in the same contexture or trans-contextural switching to another neighboring contexture. A selector as a complexion can be realized at once in different contextures. It could therefore be called a "poly-selector". Such a poly-selector can be defined as an overlapping of an intra-contextural selector "sel" and a trans-contextural selector, called *elector* "elect".

Thus, samba (elect, 1) = (sel).

The elector "elect" is (s)electing the contexture in which a selector "sel" is selecting its linearly ordered atomic terms and elements.

Selectors are acting in a pre-given order, they are conventionally introduced, not produced, but inherited from logocentric semiotics. Electors are involved in the evocation of new orders, new contextures, to give space for distributed selectors, positioning them into the graphematic matrix. Distributed selectors are constructed in the graphematic play of (s)electors and are not inherited from logocentrism.



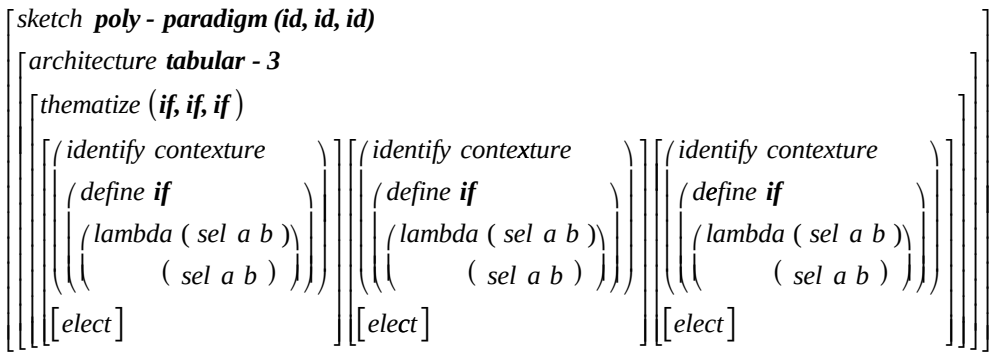
Contextural decisions are tabular with one dimension involving *selectors* and the other dimension *electors*. Each position in a polycontextural programming language is defined simul taneously by its *electors* and its *selectors*. There are no selectors without electors and there are no electors without selectors; both are designing the field of polycontextural reasoning and computing.

$$\left[\begin{array}{l} identify\ contextures^{(m)} \\ \left[\begin{array}{l} define\ \mathbf{elector} \\ \left[\begin{array}{l} lambda\ (\ contextures^{(m)}\ ) \\ \left[\begin{array}{l} elect\ contexture \\ \left(\begin{array}{l} define\ \mathbf{selector} \\ \left(\begin{array}{l} lambda\ (\ a\ b\ ) \\ \quad (\ sel\ a\ b\ ) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right]$$

This simply means, that the IF-abstraction for disseminated Ruby has an intra-contextural dimension, which is given by the operator *sel*, and a trans-contextural dimension which is introduced by the operator *elect*. Both together, as the complex operator *(s)elect*, are ruling the basic control structures of IF-THEN-ELSE in complex constellations.

A simple distribution of the IF-construct as $IF^{(3)}$=(if, if, if) is demonstrating its use. The operators of selection for *if* and the operator for selection of contextures *elect* are set in the following pattern. Thus, the abstract mechanism of the IF-THEN-ELSE-construct is distributed over 3 contextures without considering further distinctions, like condition of mediation and reflection / interaction.

Distribution of the IF-condition over 3 contextures



Ruby: if

```
Syntax:
        if expr [then]
          expr...
        [elsif expr [then]
          expr...]...
        [else
          expr...]
        end

if expressions are used for conditional execution. The
values false and nil are false, and everything else are true.
syntax.html#if
```

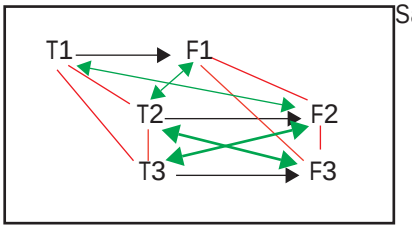
The operator *if* in Ruby is pre-defined. It is one of the reserved words. (In ARS it is derived from the operator *sel*.) Thus, it can only be used but not changed or involved into transforming interactions with other operators. The polycontextural approach of ConTeXtures, which is playing with a generalized and disseminated Lambda Calculus, gives a kind of an explicit definition of the operator *if*. This is based on the selector operator *sel* which is based on truth-constellations which again are based on *sel*. Thus, also made explicit in the programming language ARS, the introduction is involved in some circularity. This can be reflected and properly managed by a chiastic modeling of the interdependency of *sel* / *if* by a polycontextural distribution in ConTeXtures.

Comment on Implementation: mediators + compilers

It shouldn't come as a big thing to think the realization, implementation and accessibility as distributed computational systems interacting with each other and being programmed and represented by a multitude of GUIs. In the simplest case representing a multitude of windows accessible to one or more programmers at once. In a more generous setting the systems can be distributed over a network, say Internet, and *mediators* similar to compilers would have to mediate the distributed programming on mediated poly-processor computing systems. Mediators would have to parse the different programming approaches in respect to their mediability. That is, conditions of mediation would have to be checked, optimized and debugged. Thus, the chain of realization from programing to compiling has first to be augmented by a system of mediation. The new paradigm of realization now is programming-mediating-compiling in distributed and mediated programming languages.

11.3.4 Conditions of mediation

Because of the narrowness of a mediated 3-contextural system not all possible combinations of operators are fulfilling the conditions of mediation. Some hints are given below. It is not the place to give a more developed approach to mediation.



Samba is the lambda of ConTeXtures.

samba ((true, false), 3) =
[define $true_i$ ord $false_i$, i=1,2,3
define $true_1$ coinc $true_3$
define $false_1$ exch $true_2$
define $false_2$ coinc $false_3$]

$samba^{(3)}$(**id, red$_1$, red$_1$**) – *horizon*

$$\left[\begin{array}{l} thematize\ (\mathbf{true, true, true}) \\ \left[\left[\begin{array}{l} identify\ context ure_1 \\ \left(\begin{array}{l} define\ \mathbf{true_1} \\ \left(lambda\ (\ ab\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \varnothing \quad \varnothing\right] \\ \left[\left[\begin{array}{l} identify\ context ure_2 \\ \left(\begin{array}{l} define\ \mathbf{true_1} \\ \left(lambda\ (\ a\ b\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \varnothing \quad \varnothing\right] \\ \left[\left[\begin{array}{l} identify\ context ure_3 \\ \left(\begin{array}{l} define\ \mathbf{true_1} \\ \left(lambda\ (a\ b\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \varnothing \quad \varnothing\right] \end{array}\right]$$

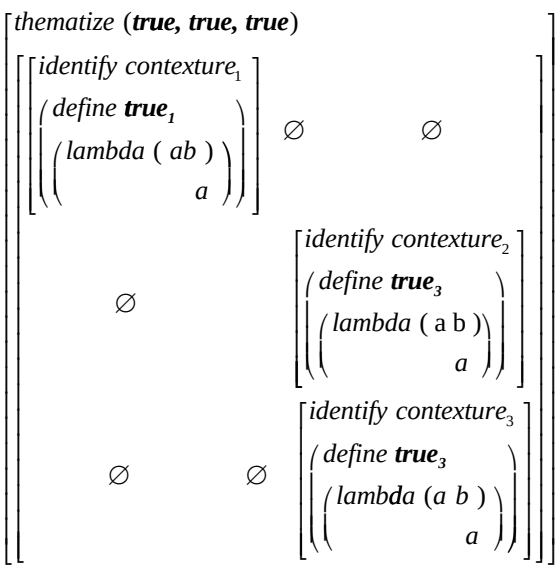
$samba^{(3)}$(**id, red$_1$, id**) – *horizon*

$$\left[\begin{array}{l} thematize\ (\mathbf{true, true, true}) \\ \left[\left[\begin{array}{l} identify\ context ure_1 \\ \left(\begin{array}{l} define\ \mathbf{true_1} \\ \left(lambda\ (\ ab\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \varnothing \quad\quad \varnothing\right] \\ \left[\left[\begin{array}{l} identify\ context ure_2 \\ \left(\begin{array}{l} define\ \mathbf{true_1} \\ \left(lambda\ (\ a\ b\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \varnothing\right] \\ \left[\varnothing \quad\quad \varnothing \left[\begin{array}{l} identify\ context ure_3 \\ \left(\begin{array}{l} define\ \mathbf{true_3} \\ \left(lambda\ (a\ b\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right]\right] \end{array}\right]$$

Scheme local true false

$$\left[\begin{array}{l} identify\ context ure_i \\ \left(\begin{array}{l} define\ \mathbf{true_i} \\ \left(lambda\ (a\ b\ )\right) \\ \qquad\qquad a \end{array}\right) \end{array}\right] \left[\begin{array}{l} identify\ context ure_i \\ \left(\begin{array}{l} define\ \mathbf{false_i} \\ \left(lambda\ (a\ b\ )\right) \\ \qquad\qquad b \end{array}\right) \end{array}\right]$$

samba$^{(3)}$ (***id, red***$_3$***, id***) – *horizon*

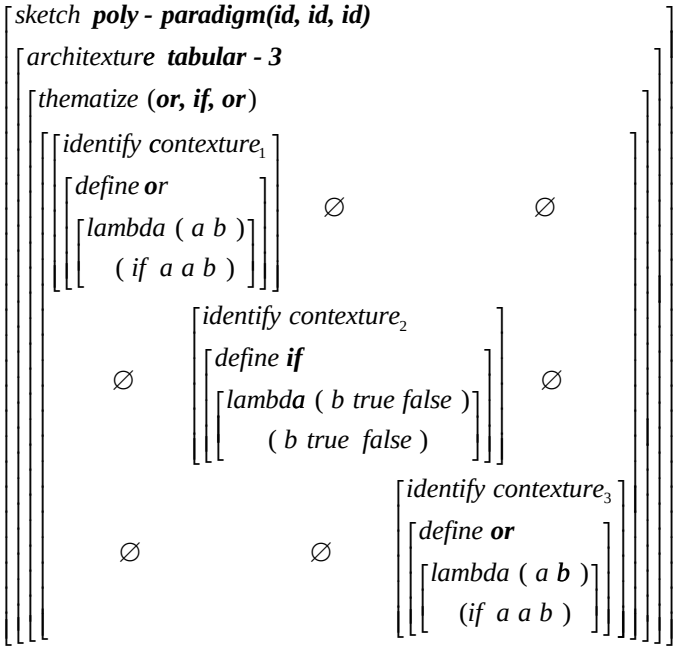


The same procedure as for the Boolean *true* has to be applied for the Boolean *false*.

A mix of the Booleans true and false don't get enough logical space in a 3-contextural setting. For m=4, negations become commutative and thus some independency is possible.

11.3.5 Violating the mediation rules

As an example of violation of the conditions of mediation the case for (or_1, if_2, or_3) is considered.



The easiest way to demonstrate the incompatibility of the combination (or, if, or) we can ask for its Boolean representation. This approach can be generalized and applied to c-obs in general delivering a method to check compatibility and realisability of mediations, esp. of poly-topic mediations.

```
( id, id, id )
( bdisp! ( or1, if2, or3 ):
( bdisp! or1 )
( bdisp! if2 )
( bdisp! or3 )

( bdisp! or1):
( bdsp! or true1 true1)  --> true1
( bdsp! or true1 false1)  --> true1
( bdsp! or false1 true1)  --> true1
( bdsp! or false1 false1)  --> false1

( bdisp! if2 ):
( bdsp! if true2 true2)  --> true2
( bdsp! if true2 false2)  --> false2
( bdsp! if false2 true2)  --> true2
( bdsp! if false2 false2)  --> true2

( bdisp! or3):
( bdsp! or true3 true3)  --> true3
( bdsp! or true3 false3)  --> true3
( bdsp! or false3 true3)  --> true3
( bdsp! or false3 false3)  --> false3
```

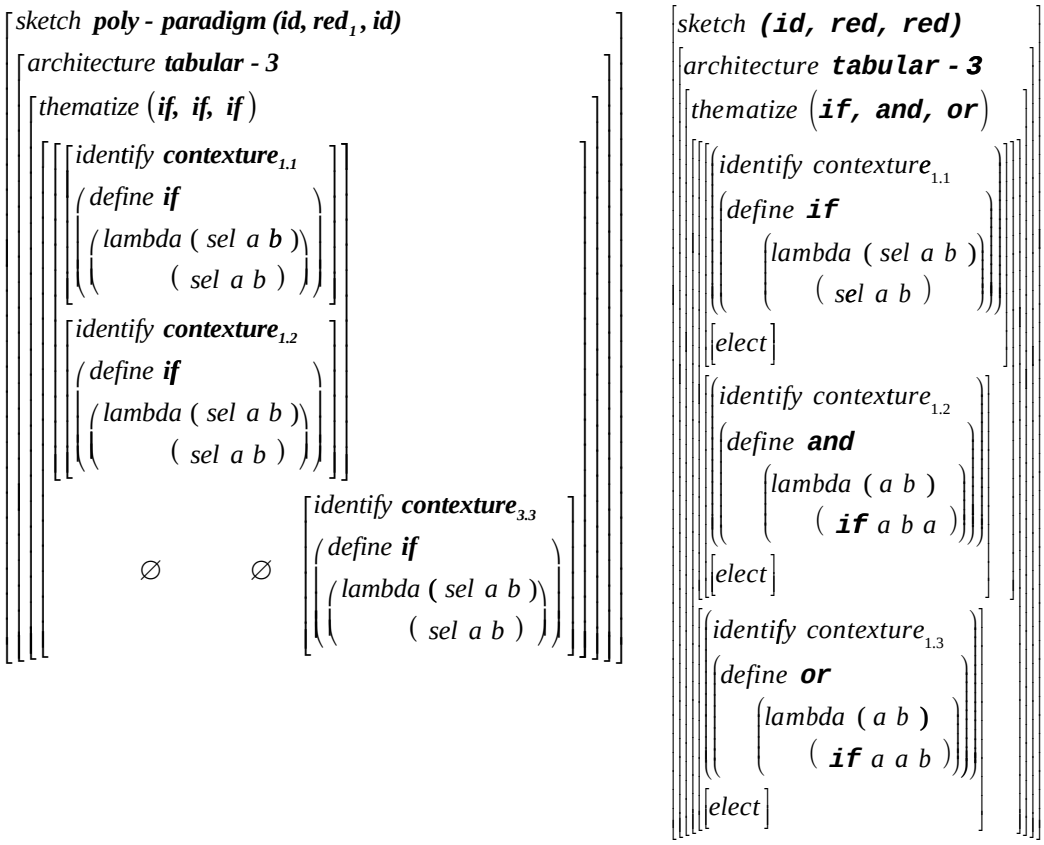
If we confront this result with the proemial conditions of our truth values, esp. $false_2$ coinc $false_3$, we observe a collision between the two results:

```
( bdsp! if2 false2 false2)  --> true2
( bdsp! or3 false3 false3)  --> false3
```

The values $true_2$/$false_3$ are neither equal nor analog, they are different and belong to the exchange relation and not the coincidence relation, like, $true_1$/$true_3$, $false_1$/$true_2$ or $false_2$/$false_3$. Thus, the mediation of the combination (or1, if2, or3) is not realized. According to the condition of mediation, based on the proemial relation of the truth values, it is enough to check the *diagonal values* of the binary constellation.

```
Thus:
( bdisp! or1)
( bdsp! or true1 true1)  --> true1
( bdsp! or false1 false1)  --> false1
( bdisp! if2 )
( bdsp! if true2 true2)  --> true2
( bdsp! if false2 false2)  --> true2
( bdisp! or3)
( bdsp! or true3 true3)  --> true3
( bdsp! or false3 false3)  --> false3
===>  ( bdsp! if false2 false2) ≠≠ ( bdsp! or false3 false3),
      ( bdsp! or true1 true1)   == ( bdsp! or true3 true3),
      ( bdsp! or false1 false1) == ( bdsp! if true2 true2).
```

Two accepted mediations of distributed if: (if1, if1, if3) and (if1, if1, if1)



Comments

The distribution of the if-construct in a 3-contextural programming language is very reduced because of its low complexity (m=3) and the necessity to fulfill the conditions of mediation. If we restrict our interest to local situations only then we can de-couple the linked systems and study the behavior of the system in focus locally. But mediation means, that we have to deal with several contextures. Thus some dynamic of local/global categories are at place. The local consideration is not reduced to one and only one contextural system. Local/global is a functional concept. Several systems can be involved and be considered local. But this makes sense only for systems of higher complexity. Thus, to study a mediated 3-contextural system demands a global approach.

The above two examples are showing how a global realization of the mediation of 3 if-constructs can be realized with the help of the super-operator reduction. That is, in the first case, contexture2 is simply mapped into the *locality* of system1. And for the second case, contextures2 and 3 are mapped into the locality of contexture1, too. It is obvious that other constructions thus are possible to produce other distributions and mediations.

Because we are still very much constructing polycontextural systems bottom-up, along the guidelines of existing systems, aspects which are independent of those classic constructs are not yet emphasized enough. That is, the interplay between reflectional/interactional computational systems will be prior to questions of truth-semantics.

11.3.6 Some if-based control structures (IF-THEN-ELSE) for Ruby(3)

$$\left[\begin{array}{l} \textbf{\textit{if - patterns}} \\ \left[\begin{array}{l} (\mathrm{if}_1, \mathrm{if}_1, \mathrm{if}_1) \\ (\mathrm{if}_1, \mathrm{if}_1, \mathrm{if}_3) \\ (\mathrm{if}_1, \mathrm{if}_3, \mathrm{if}_1) \\ (\mathrm{if}_1, \mathrm{if}_3, \mathrm{if}_3) \end{array}\right] \end{array}\right]$$

Complexions of if-structures are closely related to the distribution of their underlying logical truth-values. Thus, if-complexions are based on mediated truth-values as it was shown before. There are as many IF-THEN-ELSE structures as there are if-complexions.

There is not only one concept of the IF-THEN-ELSE construct regulating conditional situations. Different conditional if-rules of different strength and different scope of distribution are opening up a complex development of if-based control structures. With *elect* cross-over control structures or "*bridges*" are easily defined.

A distribution of IF-THEN-ELSE control structure based on $(\mathrm{if}_1, \mathrm{if}_1, \mathrm{if}_3)$ is shown below.

$$\left[\begin{array}{l} \textit{sketch } \textbf{\textit{poly - paradigm (id, red, id)}} \\ \textit{thematize } \textbf{\textit{if - control}}^{(3)}\left(\textbf{\textit{if}}_1, \textbf{\textit{if}}_1, \textbf{\textit{if}}_3\right) \\ \left[\begin{array}{ll} \left[\begin{array}{l} \textit{identify contexture}_1 \\ \left[\begin{array}{l} (\textit{define } \textbf{\textit{if - control}} \\ \left(\begin{array}{l} (\textit{lambda } (\textbf{\textit{expr stmt}}) \\ \left(\begin{array}{l} (\textit{if } \langle \textbf{\textit{boole - expression}} \rangle \\ \left(\begin{array}{l} \textit{then } \langle \textbf{\textit{true - statement}} \rangle \\ \textit{else } \langle \textbf{\textit{false - statement}} \rangle \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right] \\ [\textit{elect}] \end{array}\right] & \varnothing \quad \varnothing \\ \left[\begin{array}{l} \textit{identify contexture}_1 \\ \left[\begin{array}{l} (\textit{define } \textbf{\textit{if - control}} \\ \left(\begin{array}{l} (\textit{lambda } (\textbf{\textit{expr stmt}}) \\ \left(\begin{array}{l} (\textit{if } \langle \textbf{\textit{boole - expression}} \rangle \\ \left(\begin{array}{l} \textit{then } \langle \textbf{\textit{true - statement}} \rangle \\ \textit{else } \langle \textbf{\textit{false - statement}} \rangle \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right] \\ [\textit{elect}] \end{array}\right] & \varnothing \quad \varnothing \\ \varnothing \quad \varnothing & \left[\begin{array}{l} \textit{identify contexture}_3 \\ \left[\begin{array}{l} (\textit{define } \textbf{\textit{if - control}} \\ \left(\begin{array}{l} (\textit{lambda } (\textbf{\textit{expr stmt}}) \\ \left(\begin{array}{l} (\textit{if } \langle \textbf{\textit{boole - expression}} \rangle \\ \left(\begin{array}{l} \textit{then } \langle \textbf{\textit{true - statement}} \rangle \\ \textit{else } \langle \textbf{\textit{false - statement}} \rangle \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right] \\ [\textit{elect}] \end{array}\right] \end{array}\right] \end{array}\right]$$

Similar constructions are accessible for imperative control structures, like *while*. And again, poly-paradigmatic mixtures of control structures are well implemented. Thus, e.g. (IF-THEN-ELSE, WHILE, INHERIT) could be realized in a 3-contextural situation.

After the polycontextural matrix with its architectonics is introduced, distribution of formal systems and their constructs is not a big thing. To manage the conditions of mediation, which together with the distribution constitutes a polycontextural system is slightly more difficult and depends strongly on the decision on which level of the tectonics mediation has to be realized. An appropriate level is, in this case, truth-values. A more abstract mediation would be based on "obs" or abstract "terms" in the sense of Combinatory Logic. But the most profoundly based mediation would require an introduction of morphogrammtic and keno-grammatic structures which are neutral to the identity of disseminated truth-values or abstract objects.

11.3.7 Transjunctional commands for Ruby$^{(3)}$

$$Ruby^{(m)} : \left[Ruby^{(m)}\right]_{refl,\ act} \xrightarrow[sops]{} \left[Ruby^{(m)}\right]_{refl,\ act}$$
$$COMMAND^{(3)} : COMMAND^{(3)} \xrightarrow[\sup ob]{} COMMAND^{(3)}$$
$$id : \forall i, j \in s(m): \quad \left(Ruby^{i.j}\right) \xrightarrow[id]{} \left(Ruby^{i.j}\right)$$
$$bif\,(i,\ j) : \forall i, j \in s(m): \quad \left(Ruby^{i},\ Ruby^{j}\right) \xrightarrow[bif]{} \left(\left(Ruby^{i} \parallel Ruby^{j}\right),\ Ruby^{j}\right)$$

$$(bif,\ id,\ id)\left[COMMAND^{(3)}\right] : COMMAND^{(3)} \xrightarrow[(trans,\ and,\ and)]{} COMMAND^{(3)}$$
$$(bif_1)\left[COMMAND^{(3)}\right] : COMMAND^{(3)} \xrightarrow[trans]{} COMMAND^{1} \parallel COMMAND^{2} \parallel COMMAND^{3}$$
$$(id_2)\left[COMMAND^{(3)}\right] : COMMAND^{2} \xrightarrow[and]{} COMMAND^{2}$$
$$(id_3)\left[COMMAND^{(3)}\right] : COMMAND^{3} \xrightarrow[and]{} COMMAND^{3}$$

ConTeXtures representation of (trans, and, and)

```
samba(3)( bif, id, id )
[thematize ( trans, and, and )
identify contextures(3)
[lambda ( a(3) b(3) )
 [define trans1
  [[[ elect1.3 (if a a a)
      elect1   (if b b b)
      elect2   (if a b b)
      elect2.3 (if b a b)
      elect2.3 (if b b a)]]]]
 [identify texture2
  [define and2
   [(lambda (a b)
        (if a b a))]]]
 [identify texture3
  [define and3
   [(lambda (a b)
        (if a b a))]]]]
```

The transjunctional command structure (trans, and, and) has a more familiar representation in its logical tableaux rules. Obviously they don't exist neither in Ruby nor in any other existing programming language. Simply because they are defined not inside a contexture but between different contextures representing programming languages.

ConTeXtures: trans

Syntax:

expr$^{(m)}$ `/ / /' expr$^{(m)}$

expr$^{(m)}$ `trans' expr$^{(m}$

Example: expr$^{(3)}$ trans.and.and expr$^{(3)}$

and: as given, applied to contexture$_2$ and contexture$_3$.

trans: if left and right hand site true1, then true1for contexture1 and true1 for contexture$_3$,
if left and right hand site false, then false1 for contexture1 and true2 for contexture$_2$,
if left hand site true1 and right hand site false1 and
left hand site false1 and right hand site true1,
then false2 for contexture$_2$ and false3 for contexture$_3$.

11.4 Transjunctional constellations and tableaux proofs

Transjunctions in logic and programming are not well known. Thus, I present a proof of a formula from PolyLogics to show the functioning of a transjunctional constellation.

Step-wise concretization of the presentation. *First*, simple formulas can be written without considering the OM-structures, using only the rules of the sub-system-indices of the formulas. *Second*, especially for transjunctional formulas, the sub-system structure, O (=S), is involved but omitting the M-structure. *Third*, for full interactional and reflectional formulas, the whole OM-structure has to be used. The following diagrams shows the development of a simple negational and transjunctional formula, using only the sub-system structure. All those notational forms are for manual use only and to guide necessary further implementations. A *further* step follows by the application of meta-rules of the term calculus. *Finally*, a semi-automated proof by the LOLA-implementation is presented (in the paper mentioned below).

11.4.1 Tableaux rules for the proof of the exemplary formula H1

$$H1:\left(X\ <>\wedge\wedge\ Y\right)\rightarrow\rightarrow\ \rightarrow\ \neg_5\left(\neg_5 X\ \vee<>\vee\ \neg_5 Y\right)$$

Tableaux representation of polylogical (trans, and, and)

$$\frac{t_1\ X\ <>\wedge\wedge\ Y}{\begin{array}{c} t_1\ X \\ t_1\ Y \end{array}} \qquad \frac{f_1\ X\ <>\wedge\wedge\ Y}{\begin{array}{c} f_1\ X \\ f_1\ Y \end{array}}$$

$$\frac{t_2\ X\ <>\wedge\wedge\ Y}{\begin{array}{c|c} t_2\ X & f_1\ X \\ t_2\ Y & f_1\ Y \end{array}} \qquad \frac{f_2\ X\ <>\wedge\wedge\ Y}{\begin{array}{c|c||c|c} f_2\ X & f_2\ Y & f_1\ X & t_1\ X \\ & & t_1\ Y & f_1\ Y \end{array}}$$

$$\frac{t_3\ X\ <>\wedge\wedge\ Y}{\begin{array}{c||c} t_3\ X & t_1\ X \\ t_3\ Y & t_1\ Y \end{array}} \qquad \frac{f_3\ X\ <>\wedge\wedge\ Y}{\begin{array}{c|c||c|c} f_3\ X & f_3\ Y & f_1\ X & t_1\ X \\ & & t_1\ Y & f_1\ Y \end{array}}$$

Distribution matrix for (trans, and, and)

PM	$O1$	$O2$	$O3$
$M1$	$log1$	$log1$	$log1$
$M2$	$\varnothing$	$log2$	$\varnothing$
$M3$	$\varnothing$	$\varnothing$	$log3$

PM	$O1$	$O2$	$O3$
$M1$	$trans$	$trans$	$trans$
$M2$	$\varnothing$	and	$\varnothing$
$M3$	$\varnothing$	$\varnothing$	and

Mapping rules for (trans, and, and)

$$(<>\wedge\wedge):L^{(3)}\ *\ L^{(3)}\longrightarrow L^{(3)}:\left[L_1,(L_2\parallel L_1),\left(L_3\parallel L_1\right)\right]$$

$$\left[\begin{array}{l} Log_1:L_1\ *L_1\ \xrightarrow[transjunct\ <>]{}\ L_1:\begin{cases} f_1\ *\ t_1,\ t_1\ *\ f_1\ \rightarrow\ f_2,\ f_3 \\ t_1\ *\ t_1\ \rightarrow\ t_1,\ t_3 \\ f_1\ *\ f_1\ \rightarrow\ f_1,\ t_2 \end{cases} \\ Log_2:L_2\ *L_2\ \xrightarrow[conjunction]{}\ L_2\parallel L_1 \\ Log_3:L_3\ *L_3\ \xrightarrow[conjunction]{}\ L_3\parallel L_1 \end{array}\right.$$

Tableaux representation of polylogical (or, trans, or)

$$\frac{t_1\ X\ \vee <> \vee\ Y}{t_1\ X \;\Big|\; t_1\ Y \;\Big\|\; \begin{array}{l|l} f_2\ X & t_2\ X \\ t_2\ Y & f_2\ Y \end{array}} \qquad \frac{f_1\ X\ \vee <> \vee\ Y}{\begin{array}{l} f_1\ X \\ f_1\ Y \end{array} \;\Big\|\; \begin{array}{l} t_2\ X \\ t_2\ Y \end{array}}$$

$$\frac{t_2\ X\ \vee <> \vee\ Y}{\begin{array}{l} t_2\ X \\ t_2\ Y \end{array}} \qquad \frac{f_2\ X\ \vee <> \vee\ Y}{\begin{array}{l} f_2\ X \\ f_2\ Y \end{array}}$$

$$\frac{t_3\ X\ \vee <> \vee\ Y}{t_3\ X \;\Big|\; t_3\ Y \;\Big\|\; \begin{array}{l|l} f_2\ X & t_2\ X \\ t_2\ Y & f_2\ Y \end{array}} \qquad \frac{f_3\ X\ \vee <> \vee\ Y}{\begin{array}{l} f_3\ X \\ f_3\ Y \end{array} \;\Big\|\; \begin{array}{l} f_2\ X \\ f_2\ Y \end{array}}$$

Tableaux representation of (impl1, impl1, impl3)

$$\frac{t_1\ X\ \rightarrow\rightarrow\rightarrow\ Y}{f_1\ X \;\Big|\; t_1\ Y \;\Big\|\; f_3\ X \;\Big|\; t_3\ Y} \qquad \frac{f_1\ X\ \rightarrow\rightarrow\rightarrow\ Y}{\begin{array}{l} t_1\ X \\ f_1\ Y \end{array} \;\Big\|\; \begin{array}{l} t_2\ X \\ f_2\ Y \end{array}}$$

$$\frac{t_3\ X\ \rightarrow\rightarrow\rightarrow\ Y}{f_3\ X \;\Big|\; t_3\ Y} \qquad \frac{f_3\ X\ \rightarrow\rightarrow\rightarrow\ Y}{\begin{array}{l} t_3\ X \\ f_3\ Y \end{array}}$$

Tableaux representation of polylogical negations

$$\frac{t_1\ \left(\neg_1\ X^{(3)}\right)}{f_1\ X^{(3)}} \qquad \frac{t_2\ \left(\neg_1\ X^{(3)}\right)}{t_3\ X^{(3)}} \qquad \frac{t_3\ \left(\neg_1\ X^{(3)}\right)}{t_2\ X^{(3)}}$$

$$\frac{f_1\ \left(\neg_1\ X^{(3)}\right)}{t_1\ X^{(3)}} \qquad \frac{f_2\ \left(\neg_1\ X^{(3)}\right)}{f_3\ X^{(3)}} \qquad \frac{f_3\ \left(\neg_1\ X^{(3)}\right)}{f_2\ X^{(3)}}$$

$$\frac{t_1\ \left(\neg_2\ X^{(3)}\right)}{t_3\ X^{(3)}} \qquad \frac{t_2\ \left(\neg_2\ X^{(3)}\right)}{f_2\ X^{(3)}} \qquad \frac{t_3\ \left(\neg_2\ X^{(3)}\right)}{t_1\ X^{(3)}}$$

$$\frac{f_1\ \left(\neg_2\ X^{(3)}\right)}{f_3\ X^{(3)}} \qquad \frac{f_2\ \left(\neg_2\ X^{(3)}\right)}{t_2\ X^{(3)}} \qquad \frac{f_3\ \left(\neg_2\ X^{(3)}\right)}{f_1\ X^{(3)}}$$

$$\neg_5\ X^{(3)} \coloneqq \neg_1\ \left(\neg_2\ \left(\neg_1\ X^{(3)}\right)\right) \coloneqq \neg_2\ \left(\neg_1\ \left(\neg_2\ X^{(3)}\right)\right)$$

11.4.2 Direct formula development

Tableau presentation (negation, disjunction, transjunction, implication)

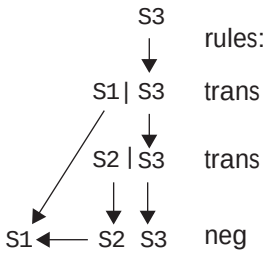
(0) f3 H1 = f3 ((X tr.et.et Y) .iij. N5 (N5 X vel.tr.vel N5 Y))

Nr	S1	S2	S3	Nr S3
1			t3 X tr.et.et Y	(0)
2			f3 N5 (N5 X vel.tr.vel N5 Y)	(0)
3	(S3)		t3 N5 X vel.tr.vel N5 Y	(2)
4	t1 X (1)		t3 X	(1)
5	t1 Y (1)	(S3)	t3 Y	(1)
6		t2 N5 X | f2 N5 X (3)	t3 N5 X | t3 N5 Y	(3)
7	(S2)	f2 N5 Y | t2 N5 Y (3)	f3 X | f3 Y	(6)
8	f1 X | t1 X (6)		x | x	
9	t1 Y | f1 Y (7)			
10	x x			

Unification

$$\frac{\alpha^{(3)}}{(\alpha^3\ \gamma^1)}$$
$$(\beta^3\ \delta^1)$$

i,j: implication
tr: transjunction
et: conjunction
vel: disjunction



$$(\varnothing, \varnothing, S3M3)$$
$$\downarrow$$
$$(S1M3, \varnothing, S3M3)$$
$$\downarrow$$
$$(\varnothing, S2M3, S3M3)$$
$$\downarrow$$
$$(S1M3, \varnothing, S3M3)$$

$$(\varnothing, \varnothing, S3)$$
$$(S1, \varnothing, S3)$$
$$(\varnothing, S2, S3)$$
$$(S1, \varnothing, S3)$$
$$(x, \varnothing, x)$$

A formula is provable in $L^{(3)}$ iff all its branches are closing under all signatures false.

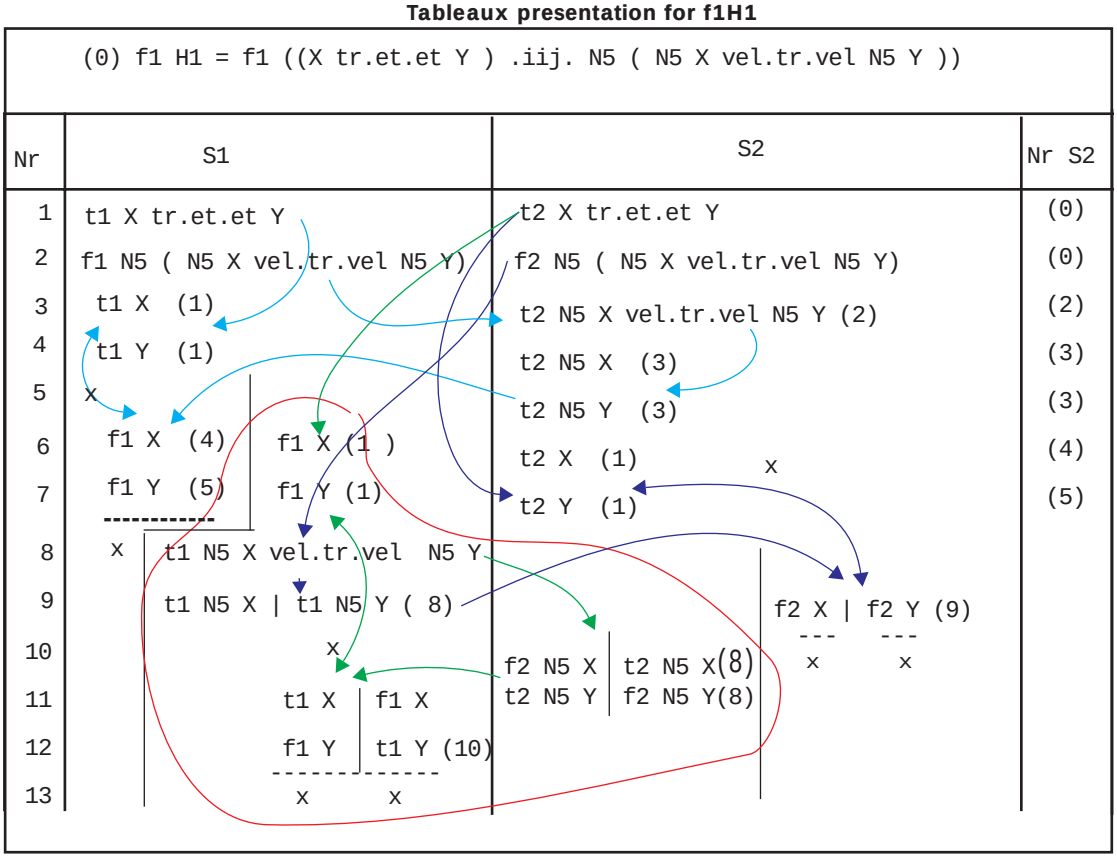
In both systems, S1 and S3, the tableaux for the formula H1 are closing under the signature of false3. For S1, a contradiction occurs between step 4/5 and 9. For S3, a contradiction occurs for step 4/5 and 7. Thus, the formula is provable under the signature false3. The same situation happens for the formula under the signature of false1. All the branches in S1 and in S2 are closing.

The signatures false1 and false3 are forced by the type of implication involved, i.e. [impl1, impl1, impl3] which is of pattern [id, red, id].

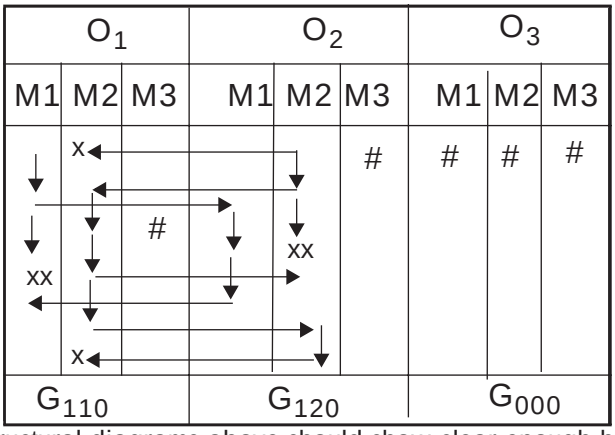
Thus, the formula H1 is true under all evaluations.

Cf. http://www.thinkartlab.com/pkl/lola/PolyLogics.pdf

Tableaux presentation for f1H1



Sub-systems S1 and S2 are closing directly, S1 at step 8 and S2 at step 9. This would be enough to close the tableaux for S1 and S2. But there is an additional part of the formula which is closing separately in S1, closing at step 13, encircled in red. All branches of the tree are closing for both signatures, thus the formula is a 3-tautology.

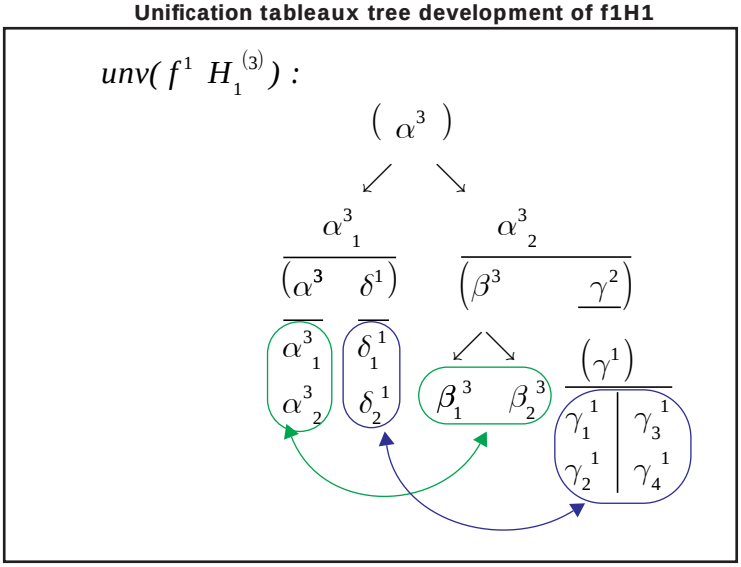


The different structural diagrams above should show clear enough how the formula development is working. The step-wise development of the formula guarantees the connectedness of the branches of the different trees, despite the jumps into other sub-systems, which are necessary to produce a semantic result on the base of the signatures.

11.4.3 Reduction of complexity by unification and meta-rules

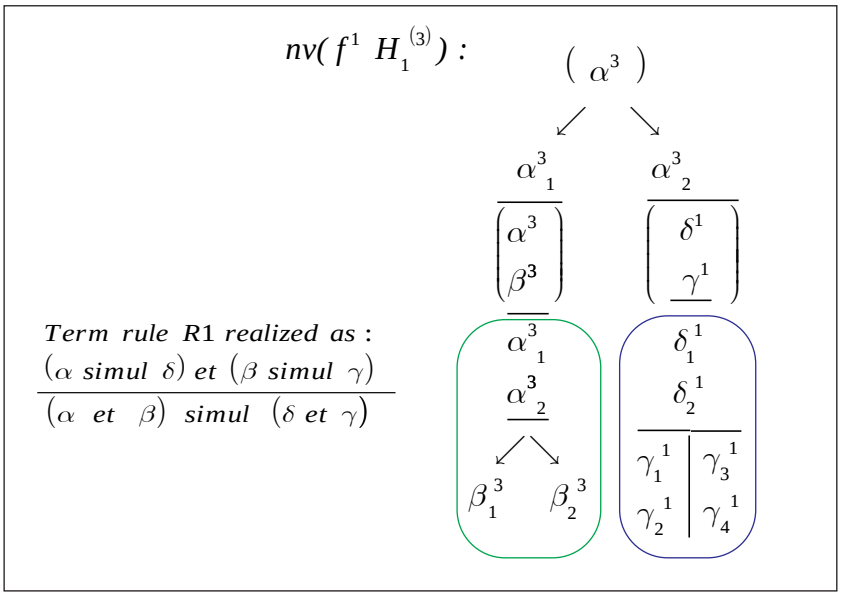
The concrete complexity of the tableaux tree of formula H1 can be reduced with the help of the unification method of Smullyan and, as a step further, with the application of some meta-rules over tableaux trees, that is, the term rules of polylogic. The diagram structure is represented by the indices of the sub-systems only.

Unification tableaux tree development of f1H1



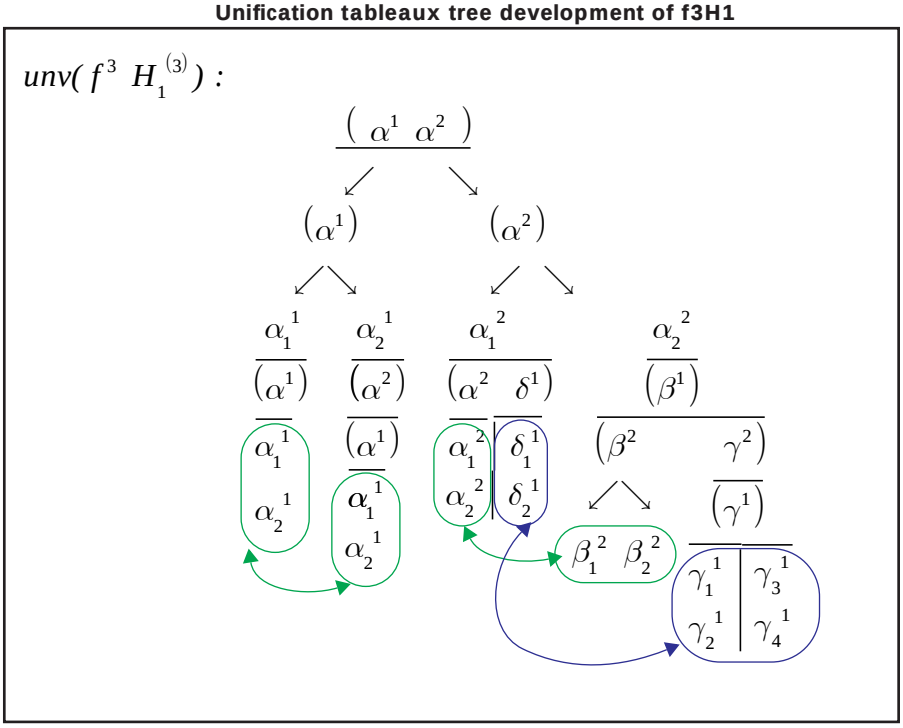
The example of unification preserves the tree structure induced by the formula. The next example, which applies additionally the term rule R1, is reducing and separating junctional and transjunctional parts, and therefore, offering a better economy of the sub-system parts which in the latter example are still distributed over the whole tree.

Unification of f1H1 with meta-rules



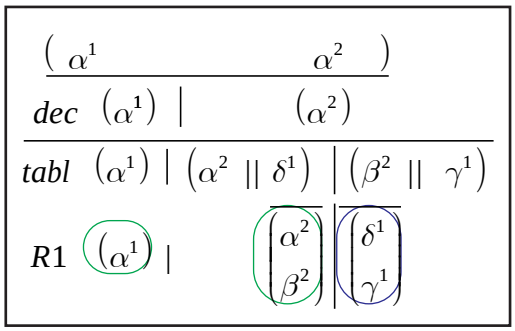
The tableau tree for f3H1 is even more confusing in its full concrete manual development, which first has to be studied before it can be reduced with the application of meta-rules to the following unification.

Unification tableaux tree development of f3H1



Again, the meta-rules are applied, producing a simple house holding of the branches and their sub-systems.

Unification of f3H1 with meta-rules



System changes, represented as a change in the index of the formula, say from gamma2 to gamma1, are produced by logical negators. Their rules are given with the tableaux rules for negations of signed formulas. Signed formulas are very convenient for complex logics but the polylogical rules are not depending on the method of signed formulas. Each method which is keeping the sub-system indices right is doing the job.

Comment:

```
dec:  decomposition of the complexion into alpha1 and alpha2 parts,
tabl: tableaux rules, producing intra-contextural sub-formulas,
R1:   term rule R1, collecting junctional and transjunctional
      parts separately.
```

An implementation for a programmed formula development for the unifyied approach would have to consider both, unification and meta-rules (cf. LOLA).

11.4.4 Function structure of Ruby(3)

Similar, functions are understood as distributed and involved in polycontextural functionality.

Classes in Ruby are first-class objects-each is an instance of class Class.

When a new class is created (typically using class Name ... end), an object of type Class is created and assigned to a global constant (Name in this case). When Name.new is called to create a new object, the new method in Class is run by default. This can be demonstrated by overriding new in Class:

```
class Class
   alias oldNew  new
   def new(*args)
     print "Creating a new ", self.name, "\n"
     oldNew(*args)
   end
 end

http://www.rubycentral.com/ref/ref_c_class.html#inherited
```

The class-definition scheme is: *"class Name ... end"*

What could be a class definition for complex, that is, ambiguous classes? They are surely excluded from the usual game. Simply because they are not members of a single hierarchic order of classes and sub classes.

Thus, a first step should be its distribution: *"class Name...end"* over different places.

$$class^{(3)} = \begin{bmatrix} \text{class Name ... end} \\ \text{class Name ... end} \\ \text{class Name ... end} \end{bmatrix}$$

A next step would be to define operations between classes of different contextures. This may be the job of operations based on polylogical operators, especially transjunctions as the interactional operators of polylogics. Operations based on the *electors* would define the orthogonal developments of classes and produce heterarchic inheritance patterns additional to the intra-contextural hierarchic orders.

General scheme for class

$$\begin{bmatrix} identify\ contexture_i \\ \left(define\ \mathbf{class}_i \right. \\ \left(\begin{array}{l} (\ lambda\ (\ \mathbf{name}\ )\ ) \\ define\ (\mathbf{constructors}) \\ define\ (\mathbf{methods}) \end{array} \right) \\ \left. end \right) \\ elect\ \ contexture_j \end{bmatrix}$$

Again, we have to chose the contextures within we want to define a *new class*. This class will have a *name* and will be defined by its *constructors* and *methods*. The introduction will be closed by *end*. *Elect* will decide in which contexture, the same or another, the next steps will happen. But this is a local introduction only not yet involved in the global interacting and reflecting game of complex class definition. Thus, this class scheme has to be disseminated and again, the super-operators *sops* will involve it into the game of interactionality / reflectionality of polycontextural programming. Class will still be involved locally in hierarchies but contextures are heterarchically organized. On the base of distributed classes over different hierarchies multiple inheritance, i.e., *poly-inheritance* is accessible without artificial restrictions. Which are necessary in the classic case to avoid contradictions.

12 Chiasms: Contradiction vs. Mediation in Polysemy

"Multiple inheritance is good, but there is no good way to do it." A. Snyder 1987

12.1 Chiasm in conceptual modeling

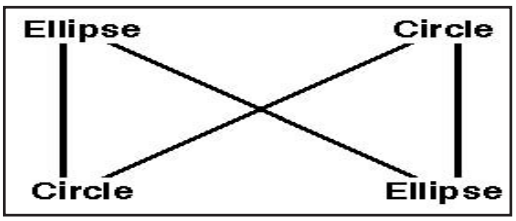


Figure 14 shows a "bowtie" inconsistency that sometimes arises in the process of aligning two ontologies. On the left of Figure 14, Circle is represented as a subtype of Ellipse, since a circle can be considered a special case of an ellipse in which both axes are equal. On the right is a representation that is sometimes used in object-oriented programming languages: Ellipse is considered a subclass of Circle, since it has more complex methods. If both ontologies were merged, the resulting hierarchy would have an inconsistency. To resolve such inconsistencies, some definitions must be changed, or some of the types must be relabeled. In most graphics systems, the mathematical definition of Circle as a subtype of Ellipse is preferred because it supports more general transformations.

http://users.bestweb.net/~sowa/ontology/ontoshar.htm#Formal

For whom are these two positions a contradiction? Where does the inconsistency appear? Where is it placed? Obviously, both positions are clean in themselves and have their reasonable advantages. Thus, the inconsistency occurs only in the mix (interaction) of both and by a mapping of it into a third general common position, say logic. What happens? The merging produces a new object which involves both different positions producing the contradiction and at the same time denies the autonomy of both.

From an interactional point of view, in contrast to an entity ontology standpoint, it is more appropriate to consider the process of merging as a process of *conflict resolution*. This type of modelling is reasonable only if we accept the relevance of the two different view-points and if both positions have their own reason to exist at once.

The above example of a *"bowtie inconsistency"* can easily modeled as a chiastic interaction between two different positions offering at least a conceptual description of the situation as introduced. Position$_i$ might be programming(PRG), Position$_j$ (OOP).

Class and subClass Scheme

```
identify contexture_i
[define class_i
(((lambda ( name )
((define (constructors))
(define (methods)))
end
(define (subClass)
(lambda ( name )
((inherit (constructors))
(inherit (methods)))
end))))]
elect contexture_j
```

Chiasm (Ellipse, Circle, PRG, OOP):

OrdRel(Ellipse1, Circle1),
ExchRel(Ellipse1, Circle2)
OrdRel(Circle2, Ellipse2)
ExchRel(Circle1, Ellipse2)
CoincRel(Ellipse1, Ellipse2)
CoincRel(Circle1, Circle2)

Pos3: The interlocking mechanism between both contextures is modeled in contexture$_l$.

In other words, the chiasm can generally be defined as between Class and subClass and its distribution over, at least, two positions. Thus:

chiasm (Class, subClass, Pos1, Pos2).

12.2 From Ruby's class definition to ConTeXtures compound notations

In prolongations of Ruby's class definition a nice modeling of compound class definitions for reflectional and interactional ConTeXtures constellations can be introduced along the "anthropomorphic" terminology of *MyClass*, *YourClass* and *OurClass*. This is a kind of a concretization of the general scheme for Class, omitting the SubClass distinction and its chiasms, as introduced before. There are no limits to extend this scenario to situations of higher complexity and complication.

Ruby class definition:

$$\left[\begin{array}{l} class\ MyClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right]$$

Short Rudy notation:

$$\left[\begin{array}{l} compound^{(3)} \\ \left[\begin{array}{l} class_1\ MyClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right]\left[\begin{array}{l} class_2\ YourClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right]\left[\begin{array}{l} class_3\ OurClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] \end{array}\right]$$

Interpretation

MyClass can be interpreted as the Class placed in the contexture in which the statement *"Circle is represented as a subtype of Ellipse"* holds and YourClass can be interpreted as the Class placed in the contexture in which the complementary statement *"Ellipse is considered a subclass of Circle"* holds. Obviously, OurClass then can be understood as the Class OurClass placed in the contexture in which the common or mediated statements about Ellipses and Circles, independently of My- and Your-position, holds. Obviously, the My / Your / Our-terminology is My-centred. This modeling is restricted to the "diagonal" components of the full 3-contextural matrix.

Complex ConTeXture notation:

$$\left[\begin{array}{l} contextures^{(3,3)} \\ \left[\begin{array}{lll} \left[\begin{array}{l} class_{1.1}\ MyClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{2.1}\ YourClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{3.1}\ OurClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] \\ \left[\begin{array}{l} class_{1.2}\ MyClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{2.2}\ YourClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{3.2}\ OurClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] \\ \left[\begin{array}{l} class_{1.3}\ MyClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{2.3}\ YourClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] & \left[\begin{array}{l} class_{3.3}\ OurClass \\ \begin{pmatrix} def \\ ... \\ end \end{pmatrix} \end{array}\right] \end{array}\right] \end{array}\right]$$

A full *reflectional* and *interactional* scenario would have to consider the mutual thematizations of the My-, Your-, and OurClass positions. Say, from My-position, Your-position is reflecting Our-position in this or that way. Thus, the above, isolated, class definitions of MyClass, YourClass and OurClass, have to be distributed over the full *polycontextural matrix* for 3 contextures.

This polycontextural matrix is presenting only the abstract scheme of 3-contextural, reflectional and interactional, class distribution. The following matrix is more detailed, but restricted, again, to the "diagonal" components of the whole matrix.

Explicit chiasm of Ellipse and Circle with Class and subClass

```
thematize (chiasm (PRG, OOP, Circle, Ellipse))
[identify contexture_i
 [chose PRG
  [define class_i
   (lambda ( Ellipse )
    [(define (constructors))
     (define (methods)      )])
   end
   (define (subClass)
    (lambda ( Circle )
     [(define (constructors))
      (define (methods)      )]))
   end
   elect Circle_j]]
 [identify contexture_j
  [chose OOP
   [define class_i
    (lambda ( Circle )
     [(define (constructors))
      (define (methods)      )])
    end
    (define (subClass)
     (lambda ( Ellipse )
      [(inherit (constructors))
       (inherit (methods)     )]))
    end
    elect Ellipse_i]]
  [identify contexture_l
   [chose PRGxOOP
    [define class_l
     (lambda ( EllipxCirc )
      [(define (constructors))
       (define (methods)      )])
     end
     (define (subClass)
      (lambda ( CircxEllip )
       [(inherit (constructors))
        (inherit (methods)     )]))
     end
     elect Circle_i,j]]]]
```

General Scheme for chiasm (Class, subClass, Pos1, Pos2).

```
thematize chiasm(Class, subClass, Pos_i, Pos_j)
[[identify contexture(Pos)_i
  [define class_i
   (lambda ( name )
    [(define (constructors)
     (define (methods)     )
    end
    (define (subClass)
     (lambda ( name )
      [(inherit (constructors)
       (inherit (methods)     )
     end
    elect contexture_j]]
 [identify contexture(Pos)_j
  [define class_j
   (lambda ( name )
    [(define (constructors)
     (define (methods)     )
    end
    (define (subClass)
     (lambda ( name )
      [(inherit (constructors)
       (inherit (methods)     )
     end
    elect contexture_i]]
 [identify contexture(Pos)_i
  [define class_i
   (lambda ( name )
    [(define (constructors)
     (define (methods)     )
    end
    (define (subClass)
     (lambda ( name )
      [(inherit (constructors)
       (inherit (methods)     )
     end
    elect contexture_i,j]]]
```

As we learnt from Minsky's Panalogy Principle: "*If you 'understand' something in only one way then you scarcely understand it at all—because when something goes wrong, you'll have no place to go"*– we should be able to run in parallel both side of the complementary problem solving strategies. If we encounter an obstacle in one approach, we can switch to the other approach to go on. But this is only the weak understanding of the situation compared to the possibility of a simultaneous approach.

More:*The circle-ellipse dilemma. Is Circle a subclass of Ellipse, or vice-versa, or neither?* http://www.cs.man.ac.uk/arch/people/j-sargeant/inheritnotes/node21.html

13 Limits of the Idea of Objects

There are objects which are not properly identifiable. At least not in a single approach. They are nevertheless not fuzzy or undetermined but over-determined. Such objects need a double approach, a double thematization and a double identification. They are not really objects but "undecidables", in more profane words, as pictures, they are also called flip-flop figures (Kippbilder) or paradoxical constructions. But they are nearly everywhere, mainly as sentences. *"Thoughts themselves are ambiguous!"* Marvin Minsky. Without a strict procedure to disambiguate sentences with the help of a chain of contexts; which are nevertheless themselves introduced by sentences; which are ambiguous, too. There is nearly nothing which is not involved in a game of channging meanings.

Gotthard Gunther writes:

> All of Man's higher forms of communication are in their inner structure equivocal to the point of being generally ambiguous. No thought - as thought - can be absolutely and unequivocally understood. Heidegger also writes about this [3]:
>
> > *"This ability to be interpreted many different ways is no protest against the rigor of those thoughts. For all true thought of an essential nature remains, and indeed, for reasons of its existence, generally ambiguous. This ambiguity is not just the remainder of a not yet achieved formal logical clarity to be properly striven for but not attained. Rather, ambiguity is the element in which thought must move in order to become rigorous."*
>
> http://www.vordenker.de/ggphilosophy/gg_identity-neg-language_biling.pdf

If we understand "class" or "object" as a cognitive category, i.e. as a *Reflexionsbestimmung*", and not as an ontological category, then "class" and "object" are embedded in a reflectional grid reflecting on the categories and their environments.

The question leads back to the main question: Does the class, object, instance, prototype, clone model represent properly the modeled objects in question?

A pleasant new candidate for equivocal and ambiguous interactions might well be the couple *object/aspect*.

To deconstruct programming paradigms we have to discover their primary dichotomous structure, their system of couples of concepts, like class/instance, private/public, reserved/derived, object/aspect, and to diamondize, step by step, the system/network of those dichotomies towards a game of chiasms between the disseminated dichotomies. Networks of disseminated dichotomies are building a heterarchic scenario.

13.1 Why linearizations? Some citations

To avoid contradictions by involving the diamonds of polysemy the strategy of linearization was introduced.

> Kim Barrett et al, A Monotonic Superclass Linearization for Dylan
> In a class-based object-oriented language, objects are instances of classes. The properties of an object - what slots or instance variables it has, which methods are applicable to it - are determined by its class. A new class is defined as the subclass of some pre-existing classes (its superclasses - in a single-inheritance language, only one direct superclass is allowed), and it inherits the properties of the superclasses, unless those properties are overridden in the new class. Typically, circular superclass relationships are prohibited, so a hierarchy (or heterarchy, in the case of multiple inheritance) of classes may be modeled as a directed acyclic graph with ordered edges. Nodes correspond to classes, and edges point to superclasses. Languages that use this model include Ada 95, C++, CLOS, Dylan, Eiffel, Java, Oberon-2, Sather, and Smalltalk.

In object-oriented systems with multiple inheritance, some mechanism must be used for resolving conflicts when inheriting different definitions of the same property from multiple superclasses.

Some languages require manual resolution by the programmer, with mechanisms such as explicit delegation in C++ [ES 90] and feature renaming in Eiffel. [Meyer 88]

When a class is created, a linearization of its superclasses, including itself, (also known as the class precedence list or CPL) is determined, ***ordered from most specific to least specific****. When several methods are applicable for a given call, the one defined on the most specific class, according to the linearization, is selected.*

Most object-oriented languages implicitly use a rule similar to linearization for method dispatching in single inheritance: a class is more specific than any of its superclasses, so methods defined for subclasses override methods defined for superclasses. The problem with generalizing this to multiple inheritance is that the simple rule does not make clear which of two superclasses with no subclass/superclass relationship between them is more specific.

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html

famous diamond problem

How does Java solve the famous diamond problem (caused by multiple inheritance)? That is, you can implement multiple interfaces, that in turn can be implemented by one class. I know that Java has done it; my question is how?

The quick answer: Java solves the diamond problem inherit in multiple inheritance by not allowing multiple inheritance in the first place. However, as you point out, interfaces do allow a certain kind of multiple inheritance.

So what's the solution? If you have access to the source code the best course of action is to rename the methods in either of the above cases. If you don't have access, you are stuck.

http://www.javaworld.com/javaworld/javaqa/2001-03/02-qa-0323-diamond.html

Ruby: Inheritance and Mixins

Some object-oriented languages (notably C++) support multiple inheritance, where a class can have more than one immediate parent, inheriting functionality from each. Although powerful, this technique can be dangerous, as the inheritance hierarchy can become ambiguous.

Other languages, such as Java, support single inheritance. Here, a class can have only one immediate parent. Although cleaner (and easier to implement), single inheritance also has drawbacks---in the real world things often inherit attributes from multiple sources (a ball is both a bouncing thing and a spherical thing, for example).

Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
A Ruby class can have only one direct parent, and so Ruby is a single-inheritance language. However, Ruby classes can include the functionality of any number of **mixins** (a mixin is like a partial class definition). This provides a controlled multiple-inheritance-like capability with none of the drawbacks.
http://www.rubycentral.com/book/tut_classes.html

Heterarchy vs. hierarchy in Smalltalk

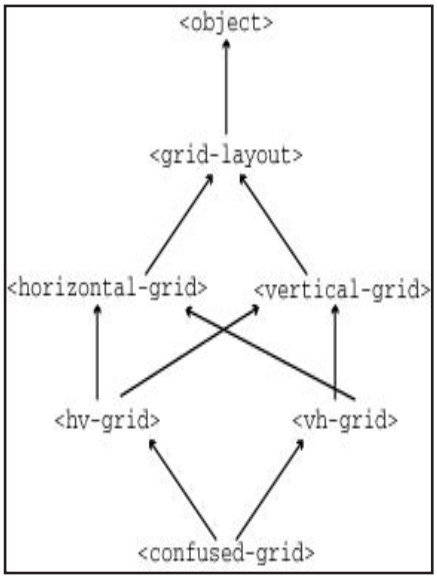
At the top of the **heterarchy** are root interfaces, which are parentless interfaces; they extend no other interfaces. At the bottom of the heterarchy are the leaf interfaces which are childless interfaces; no other interfaces extend them.
http://www.iam.unibe.ch/~scg/Archive/Papers/Sade02aDynamicInterfaces.pdf
http://www.jot.fm/issues/issue_2002_05/article1

13.2 Paradox by design and paradox by construction

Sowa's example of Circle/Ellipse was something like paradox based on the modeling of facts. It is a fact that there exists, at least, two different approaches to understand the relation between the concepts circle and ellipse. And both understandings of the conceptualization are excluding each other on logical grounds.

On the other hand, the example "confused-grid", can be understood as a constructed constellation. The design of the confused grid may have been constructed for didactical reasons but it might be in conflict with the underlying logic of class constructions. That is, the construction is over-determined in relation to the logic of class involved. It turns out that the complexity of the constructed object "confused grid" is higher than the complexity of the logic and ontology of the used programming language. The example may therefore be artificial, but it is nevertheless of interest. It demonstrates very clear the problematics of polysemy and multiple inheritance in OOP. Additionally it shows the conflict between possible constructions by design and violation of the logical base of the system in which the construction happens. But disallowance of such conflictive constructions wouldn't help much because there are enough real world applications where they appear by necessity.

13.2.1 The confused-grid example




```
define class <grid-layout> (<object>) end;
define class <horizontal-grid> (<grid-layout>) end;
define class <vertical-grid> (<grid-layout>) end;
define class <hv-grid>
        (<horizontal-grid>, <vertical-grid>)  end;
define method starting-edge
          (grid :: <horizontal-grid>)
   #"left"
   end method starting-edge;
 define method starting-edge
   (grid :: <vertical-grid>)
  #"top"
end method starting-edge;
```

Example 1a: A simple use of multiple inheritance

```
 define class <vh-grid>
        (<vertical-grid>, <horizontal-grid>) end;
```

Example 1b: Reversing classes in the linearization

```
define class <confused-grid> (<hv-grid>, <vh-grid>) end;
```

Example 1c: An inconsistent class definition

For example, consider the simple use of multiple inheritance in example 1a.

The question multiple inheritance raises is ``What is the starting-edge for an <hv-grid>?'' If it is more *like a horizontal than a vertical grid, it is the left edge, but if it is* more *like a vertical grid, it is the top edge.*

In an explicit resolution system, the author of the <hv-grid> class would have to write a declaration or method to choose which superclass to obtain the starting-edge behavior from.

In contrast, when a linearization *is used, the default behavior is determined by which of <horizontal-grid> or <vertical-grid> appears first in the linearization.*

13.2.2 Linearization of the confused-grid

Following CLOS, Dylan uses the local precedence order - the order of the direct superclasses given in the class definition - in computing the ***linearization****, with earlier superclasses considered more specific than later ones. Therefore, since <horizontal-grid>* precedes *<vertical-grid> in the definition of <hv-grid> it will also precede it in the linearization. The full linearization for <hv-grid> is*

<hv-grid>, <horizontal-grid>, <vertical-grid>, <grid-layout>, <object>

On the other hand, to create a combined horizontal and vertical grid which is more like a vertical grid than a horizontal one, the only change necessary to the definitions above would be to reverse the order of the direct superclasses in the class that combines the two grids; see example 1b.

It is possible that an inheritance graph is inconsistent under a given linearization mechanism. This means that the linearization is over-constrained and thus does not exist for the given inheritance structure.

An example of an inconsistent inheritance relationship appears in example 1c. <confused-grid> is ***inconsistent*** *because it attempts to create a linearization that has <horizontal-grid> before <vertical-grid>, because it subclasses <hv-grid>, and <vertical-grid> before <horizontal-grid>, because it subclasses <vh-grid>. Clearly, both of these constraints cannot be obeyed in the same class.*

Kim Barrett et al, A Monotonic Superclass Linearization for Dylan

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html

$$\text{<confused-grid>} \Rightarrow \begin{bmatrix}\text{<hv-grid>}\\ \text{<vh-grid>}\end{bmatrix} \Rightarrow \begin{bmatrix}\begin{bmatrix}\text{<horizontal-grid>}\\ \text{<vertical-grid>}\end{bmatrix}\\ \begin{bmatrix}\text{<vertical-grid>}\\ \text{<horizontal-grid>}\end{bmatrix}\end{bmatrix} \Rightarrow \text{<grid-layout>} \Rightarrow \text{<object>}$$

13.3 Modeling the main conflict

Obviously, the main conflict is between the aim of the construction, that is to construct a "confused-grid" and the underlying logic of the class paradigm. In fact, there is no reason to design the top/left or the top/right decision.

There is no reason from the point of view of the design to accept the simultaneity of the <hv-grid> and the <vh-grid>. But this is not only contradicting linearization but the logic of classes. There is no logical operator for class simultaneity in OOP and its logics. Such a modeling is given by the next programming diagram.

Because the process of construction was realizing some dynamics of changing viewpoint the dynamics itself has to be modeled in a full mapping of the concept construction.

Scheme of a simultaneous mapping

$$S^1\text{: <horizontal-grid>} \Rightarrow \text{<grid-layout>} \Rightarrow \text{<object>}$$

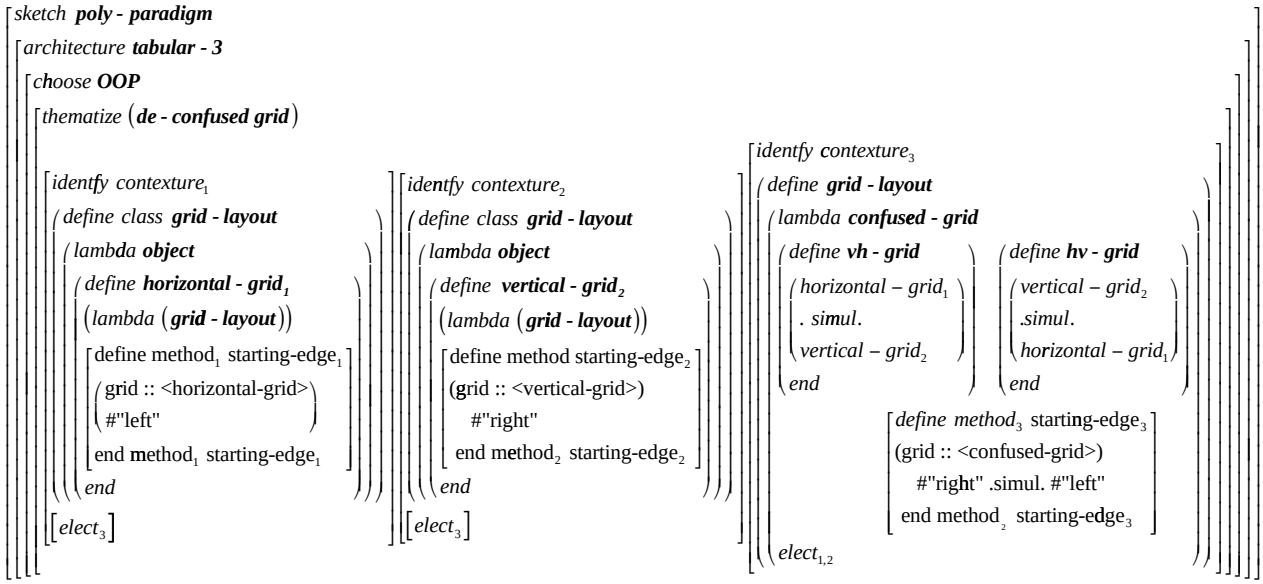
$$S^2\text{: <vertical-grid>} \Rightarrow \text{<grid-layout>} \Rightarrow \text{<object>}$$

$$S^3\text{: <hv-grid>} \Rightarrow \begin{bmatrix}\text{<horizontal-grid>}\\ \text{<vertical-grid>}\end{bmatrix} \Rightarrow \text{<grid-layout>} \Rightarrow \text{<object>}$$

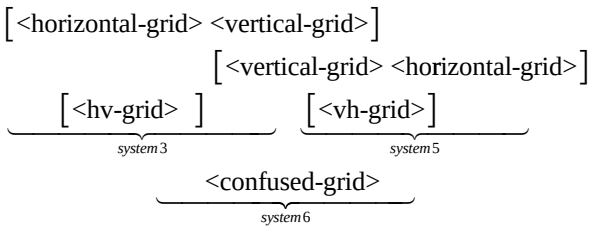
$$S^3\text{: <vh-grid>} \Rightarrow \begin{bmatrix}\text{<vertical-grid>}\\ \text{<horizontal-grid>}\end{bmatrix} \Rightarrow \text{<grid-layout>} \Rightarrow \text{<object>}$$

With that, I am modeling the confused grid as a paradox figure, a Kippbild/Vexierbild, changing from right to left and from left to right and knowing that it is an interlocking mechanism and not a succession of two separated entities. A full description, modeling and logification of this dynamics of change would involve at least 6 different contextures to be realized adequately.

13.3.1 Modeling the confused-grid as a simultaneity



13.3.2 Scheme of the confused-grid in its dynamics

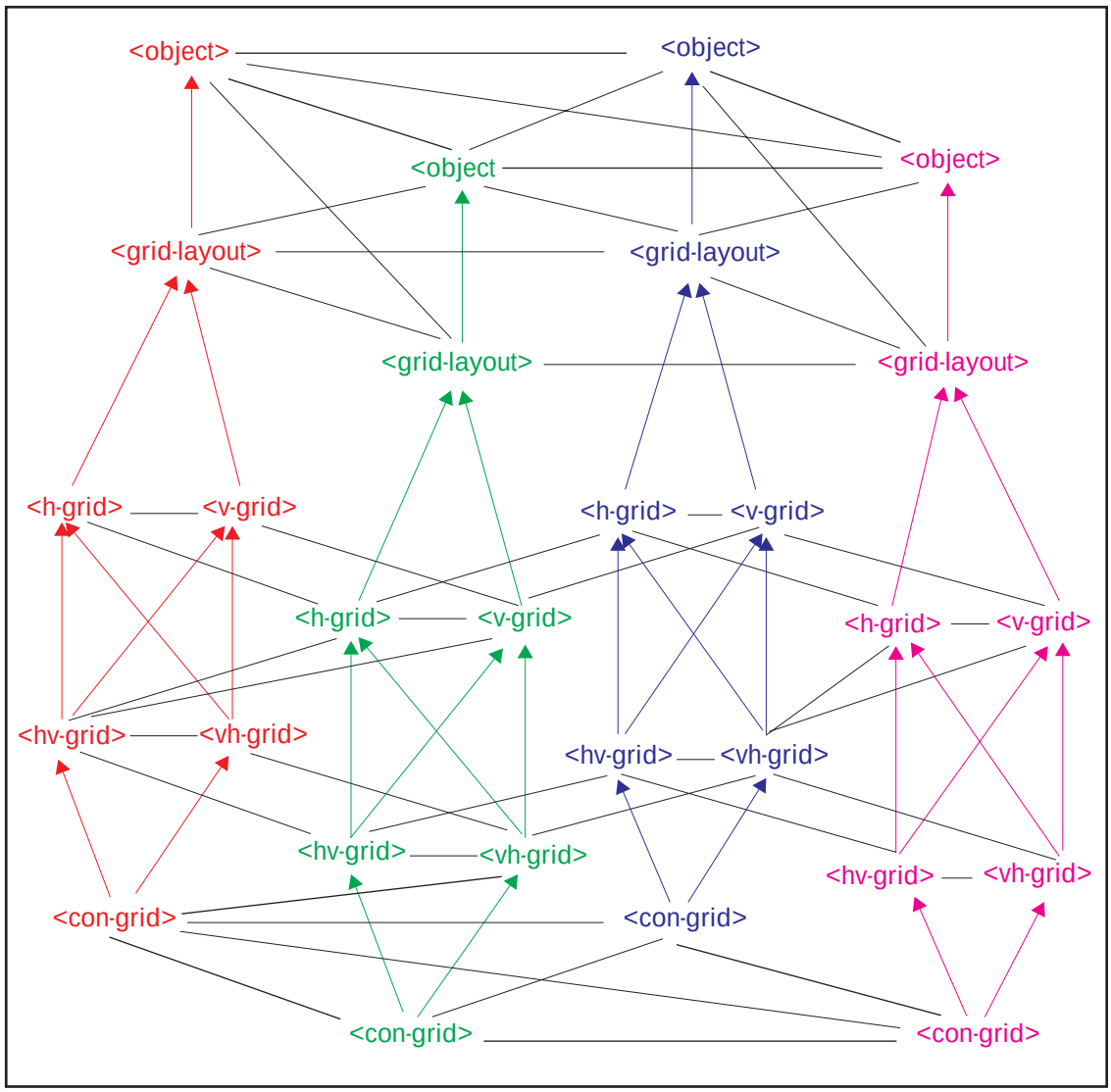


Dynamics of the paradox

1. Thematize: <horizontal-grid>
2. Thematize: <vertical-grid>
3. Observe switch of 1. and 2. : <hv-grid>
4. Thematize the inversion of switch 3. :
   4.1: <vertical-grid> 4.2: <horizontal-grid>
5. Observe switch of 4.1 and 4.2: <vh-grid>
6. Observe switch of switches 3. and 5.:
   <hv-grid> .simul. <vh-grid> = <confused-grid>

The dynamics of the scheme are not involved in any information processing in time / space and feed-back loops. But only in the structural pattern of the dynamics as change. That is, change as a switch from one contexture or viewpoint the another and back. But a switch is neutral to its beginning, thus, for an observer, it has to be inscribed in its neutrality. That's the reason for the repetition of the inverse order of the start configuration at 4. This change needs a localization in the game to be realized.

13.3.3 A more explicit visualization of the paradox; on demand



How to read the diagram depends very much on the kind of modeling involved.

To solve a problem, we have to multiply it. (Lenin) Each colored graph has its own paradox by construction. Each paradox can be resolved by bridging to another colored graph by analogy (sameness), ruled by the as-abstraction, avoiding collision.

For example <hv-grid> as green to <hv-grid> as red or/and as blue, simultaneously. Thus, avoiding paradox. Because the super class *object* is in itself a complexion of mediated different approaches ruled by a polycontextural class logic. On the other hand, because now, paradoxes are 'domesticated', all sorts of new paradoxes are accepted. Hence, of the many possibilities, the following path isn't producing any contradiction:

> <con-grid>red –> <hv-grid>red||<vh-grid>green –> <h-grid>red||<v-grid>green –> <grid-layout>red||<grid-layout>green –> <object>red||<object>green ==> no contradiction; but separation/mediation of two analogous/simultaneous ways of thematizing <con-grid>. ("–>": class inclusion, "||": discontextural parallelism).

13.4 Dialectics of linearization, evolution and mediation

$$\textit{Circular Design}: \begin{bmatrix} [\text{object}]^{(m)} \Rightarrow [\text{paradox}]^{(m)} \\ \Updownarrow \qquad\qquad \Updownarrow \\ [\text{paradox}]^{(n)} \Rightarrow [\text{object}]^{(n)} \end{bmatrix}$$

If there is no negotiation about linearization possible and the fact of the arising antagonism has to be accepted by the system and nevertheless the antagonism is not accepted as a working scenario by the designer or the main program then a resolution is opened up by a structural expansion, enlargement, augmentation of the complexity of the framework in which the conflict happens. This is a kind of a second order negotiation where the fundaments of the conflict are resolved by enlargement of the system. And it has strictly to be contrasted from the action of negotiation for renaming of terms and relations inside a framework.

Paradoxical constellations with multiple inheritance, polysemy and diamond properties are defining a problem. To solve the problem it is not wrong to consider its construction. What we can observe is the involvement of different approaches, points of view and contexts mapped into a single non-ambiguous conceptual framework. Thus, the dynamics which are used to construct the paradox are frozen in contradiction. Obviously, mono-contextural approaches don't have space to map the logical dynamics of the construction. Thus, a first resolution of the problem lies in the acceptance of the mechanism of its construction. Then, a polycontextural modeling and programming is offering space for a dynamic realization of the over-determined construction without restoring its paradox result. But, he who likes paradoxes is encouraged to build on the base of the newly introduced complexity of the system again some more complex ones.

Harmonizing complexity

But this is not the end of the story. It is always possible to construct an object which is too complex to fit into the language it belongs–or not. Thus, there is no final re-solution. A construction always can be too complex and the language not prepared to deal with it. On the other hand, a programming language can be much too over-determined, and being much too complex for the object domain it is modeling and producing unnecessary redundant patterns.

With that, polycontextural programming languages are confronted with a new challenge of optimization: to harmonize the complexity of the language and the complexity of the domain of computation. Programming as conceptual modeling and programming as computation are in a competitive interplay.

Conflictive situations can be analyzed and adjusted manually or by meta-programs.

Some background

Polycontextural activities as interactionality and reflectionality as I am trying to realize are in the tradition of the work of the cybernetician and philosopher Gotthard Gunther, inspired by the dialogical and conversational approaches and cybernetic realizations of Gordon Pask. Lively conversions with Heinz von Foerster and conversations with Humberto Maturana. Impressed by the scientific rigor of Lars Löfgren and Francisco Varela. The work of Gordon Pask has a cool representation and is kept well alive by the cybernetician and artist Paul Pangaro. To mention only a few but distinct roots of the highly complex rhizomes.
http://cyberneticians.com/ http://www.pangaro.com/
http://www.thinkartlab.com/pkl/archive/Cyberphilosophy.pdf

14 Chiasm of Deliberating Self-modification for Ruby(3)

Why should we use polycontextural paradigms?

Programmers are well used to give away some words from their language. It seems not to be considered as a serious restriction or even loss of expressivity of their programming abilities. All existing programming languages are based on some "*reserved words*". For Ruby the reserved words are listed below.

Reserved words in Ruby

BEGIN	class	ensure	nil	self	when
END	def	false	not	super	while
alias	defined	for	or	then	yield
and	do	if	redo	true	
begin	else	in	rescue	undef	
break	elsif	module	retry	unless	
case	end	next	return	until	

Now, people are talking and programming a lot of and about self-modifying systems. Especially in AI and AL there is an obsession for self-referential constructions. Everybody who ever has programmed a self-replicative program has an idea about it.

This situation, Ruby/Identifiers, can be generalized and studied in a more formal setting as done by Melvin Fitting. Questions of a *"minimal set"* of reserved terms are not in place. Here I simply want to emphasize the rules of the game.

Melvin Fitting's work space

Following the exposition of EFS (Elementary Formal Systems) by Melvin Fitting (in the tradition of Raymond Smullyan) a simple connection to the paradigm of ConTeXtures can be established by introducing EFS for each agent of a societal compound or community. Dissemination of EFS is realized along the strategies of polycontexturality as a complementary system of reflectionality and interactivity of distributed and mediated contextures giving place and place-ing Elementary Formal Systems (EFS_i). Each agent has his own work-space W and his reserved Identifiers ID based on its own syntax.

The interest of this little sketch is not to develop a full theory of disseminated EFS but to give the smallest account to make clear the possibility of chiastic self-referentiality on the base of very elementary definitions of EFS, reduced to nearly nothing but enough for dissemination, reflectionality and interactivity.

Fixed-points vs. chiasms

Obviously, the chiasm between work-space and identifier is based on the as-abstraction. Identifier as work-space, work-space as identifier, and again, identifier as identifier and work-space as work-space. Without this chiastic dynamics, restricted to the identity principle, self-referentiality is damned to useless speculations about circularity.

In contrast to circularity concepts based on recursion and fixed-points or re-entry the chiastic concepts are finite in their definition. Fixed-point modeling of self-referentiality are often connected with transfinite recursion. It is not understood, say by Second-order Cybernetics, that this might be a reasonable approach to conceptual description, it isn't a feasible and constructive approach for programming at all.

The chiastic approach, based on the proemial relation, is not only finite, constructive and programmable but part of a *construction* language in contrast to a *description* language. But it has its costs, too. We have to give up the uniqueness and homogeneity of our mono-contextural notions. A discontextural gap between heterogeneous positions has to be accepted and to be bridged by transcontextural mediation. More at:

http://www.thinkartlab.com/lola/poly-Lambda_Calculus.pdf

Syntax of the language EFS(str(L))

The data structure (sorts) of EFS(str(L)) are character strings over an alphabet.
The *alphabet* consists of a non-empty set of symbols (letters). *Words* are defined over the alphabet by the relation of concatenation. $\boldsymbol{CON_L}$ is the 3-place relation consisting of all ordered triples <w1, w2, w3> where each of w1, w2 and w3 are words over L, and w3 is the result of concatenating w1 and w2. **str(L)** is the data structure <L*, CONL>, where L is an alphabet. The EFS language for this data structure is denoted EFS(str(L)).
An *identifier* is any string made up of letters from "standard" alphabet, A, B,..., Z of capital letters together with the "brak" symbol, '_', and that does not begin or end with '_'.
Variables are v, v', v'', ... The punctuation symbols are arrow –>, parenthesis (,) and comma ,.
Terms of EFS(**str(L)**) are words of L* and variables.
Definition. If t1, ..., tn are terms, and IDENT is an identifier, then IDENT(t1, ..., tn) is an *atomic statement*.
Definition. If S1, S2, ..., Sn is a list of atomic statements, and T is an atomic statement, then S1 –> S2 –>...–> Sn –> T is a *statement*.
Definition. For an atomic statement T, we say T is in the *assigned position* in the statement S1 –> S2 –>...–> Sn –> T, and also in the unconditional statement T itself.
Definition. A *work space* **W** consists of
(1) A specification domain, the objects we are talking about.
(2) A list of identifiers, designed as reserved.
(3) A specification of what relations the reserved identifiers represent.
Relations represented by the reserved identifiers are the given relations of **W**.
The *basic work space* of EFS(**str(L)**) is:
(1) the domain L*, all words over the alphabet L.
(2) with the only reserved identifier CON.
(3) CON represents the concatenation relation on L, $\boldsymbol{CON_L}$.
Definition. We call a procedure statement *acceptable* (in a work space) if no reserved identifier occurs in the assignment position.
Definition. Given a work space **W**. A *procedure in* **W** consists of two parts: a header and a body.
The *header* of the procedure designates an unreserved identifier and a number.
The *body* of a procedure is any finite collection of statements that are acceptable in the work space **W**.
Definition. Let **W** be a work space. Suppose we write a procedure in this work space **W**, say
NAME (n):
(body of procedure **NAME**)."

Melvin Fitting, Computability Theory, Semantics, and Logic Programming,1987

Chiasm of reserved and produced words

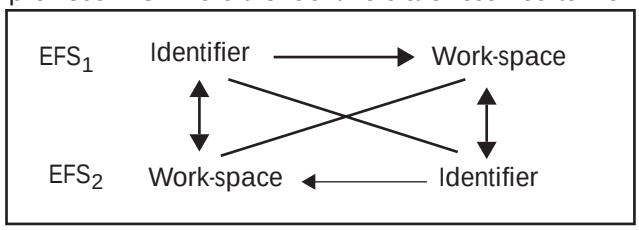
The general practice is to start with a basic work space, then to enlarge it.

> *"In the work space we start with, the relations represented by reserved identifiers can be assumed to be "given". (...) Then, starting with the basic work space, we write procedure after procedure each one characterizing some additional set or relation. When we have thus characterized a new relation, we can designate an identifier to represent it, as a new identifier. The procedure we have written constitutes the specification of what this identifier represents. In this way we produce a richer work space."*

The main dichotomy of an EFS thus is established between the given reserved *identifiers* and the accepted *statements* of the work space. No accepted statement as a program thus can change or modify the set of identifiers. Identifiers can only be introduced by the designer of the EFS and this is at least in principle a human being and not a program. And who ever is changing the set of pre-given words can do this only "outside" the running system. There is no limit in doing that.

Reserved words can be added, eliminated, their implementation can be optimized, and so on. But not while running. Simply not while running the system because its "running" is enabled exactly by the reserved words as its condition. Therefore a mono-contextural programming system is not able for principle reasons to change its own definition. It is not able to change while running its list of reserved words. And a re-implementation of those reserved words while defining the running system is impossible by definition, too. Thus, self-modification in an essential sense is not possible by definition in the framework of EFS.

But this is exactly what we are looking for. What we need is essential self-modification. Obviously, what we need is a contexture where the given identifiers are characterized as parts of accepted statements, not involved in an endless hierarchy of meta-languages but in a finite "circular" chiastic, that is, heterarchic distribution, co-existing at once with the previous EFS where the identifiers are reserved terms.



Thus, I am not looking for a "Münchhausen" solution but for a solution of essential self-referentiality, not on the base of the identity of the terms, but on the base of their sameness. This allows a distribution of the same construction (identifier, work-space, statements) over different contextural loci. And what is thematized as identifier of a work-space in one contexture acts as a statement of another work-space based on other or similar identifiers. Between identifiers and work-spaces a chiasm, distributed over different contextures, is installed. Identifiers and work-spaces occur in different contextures representing different use of the terms identifier and work-space (statements).

With that in mind, a self-modification of a programming system, that is a modification of its reserved words while running, has lost its magical circularity and has become part of polycontextural programming. *That is, poly-paradigms and chiastic interactions between paradigms, enabling each other in their togetherness, are the new features.*

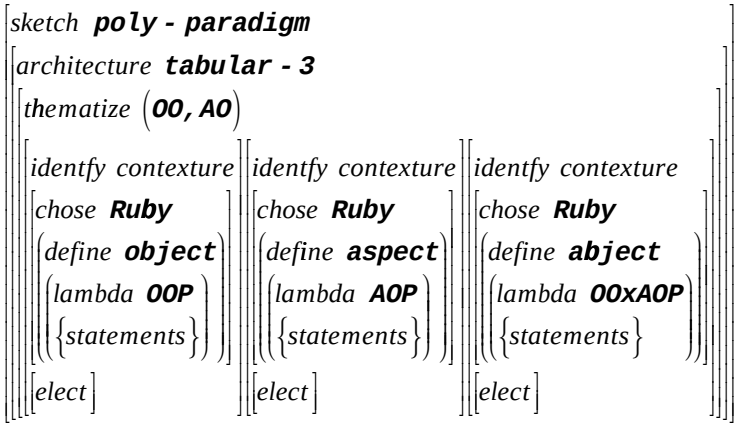
Modeling the chiasm between work-space and identifier

$$\left[\begin{array}{l} thematize\ \textbf{chiasm}\left(\textbf{work - space, identifier}\right)^{(3)} \\ \left[\begin{array}{l} choose\ \textbf{Ruby}^{(3)} \\ \left[\begin{array}{l} contextures^{(3)} \\ \left[\left[\begin{array}{l} identify\ \textbf{work - space}^{1.1} \\ \left(\begin{array}{l} lambda\ \left(\textbf{identifier}\right) \\ \left\{list\right\} \end{array}\right) \\ \left[elect\ \textbf{work - space}^{2.1}\right] \end{array}\right] \left[\begin{array}{l} identify\ \textbf{work - space}^{1.2} \\ \left(\begin{array}{l} lambda\ \left(\textbf{identifier}\right) \\ \left\{list\right\} \end{array}\right) \end{array}\right]\right] \left[\left[\begin{array}{l} identify\ \textbf{work - space}^{2.1} \\ \left(\begin{array}{l} lambda\ \left(\textbf{identifier}\right) \\ \left\{list\right\} \end{array}\right) \end{array}\right] \left[\begin{array}{l} identify\ \textbf{work - space}^{2.2} \\ \left(\begin{array}{l} lambda\ \left(\textbf{identifier}\right) \\ \left\{list\right\} \end{array}\right) \\ \left[elect\ \textbf{work - space}^{1.2}\right] \end{array}\right]\right] \left[\begin{array}{l} identify\ \textbf{work - space}^{3.3} \\ \left(\begin{array}{l} lambda\ \left(\textbf{identifier}\right) \\ \left\{list\right\} \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right]$$

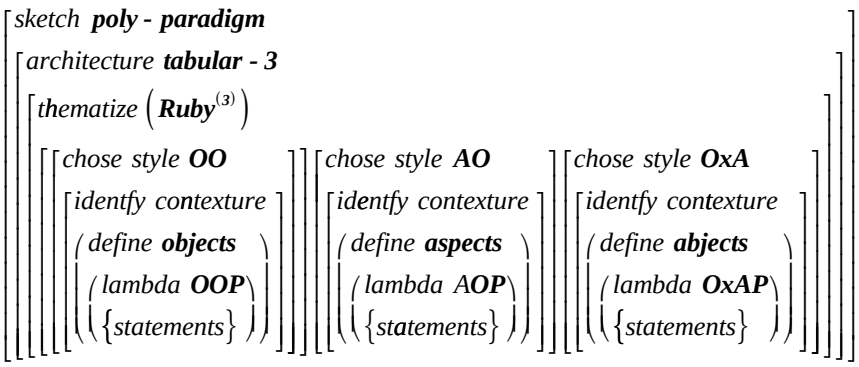
15 Distribution of Ruby as AOP and AOP as Ruby

Different interpretations of the relations between Ruby as a programming language and OOP/AOP as programming paradigms are possible. And without surprise, some combinations of both kind of modeling are stepping onto the arena, too.

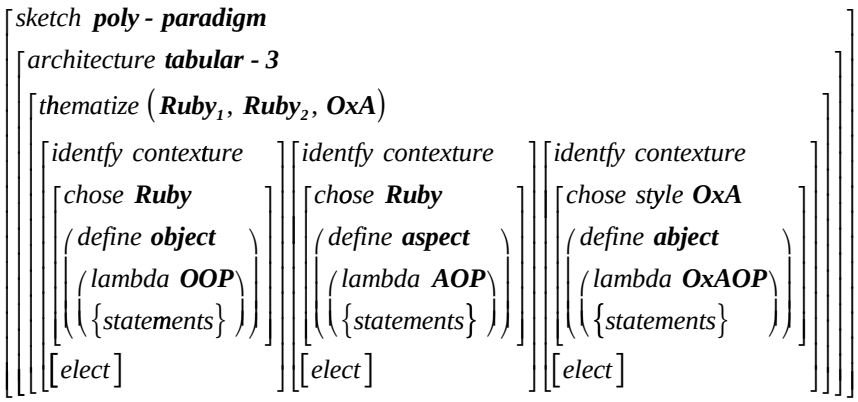
OOP and AOP and OOxAOP as Ruby



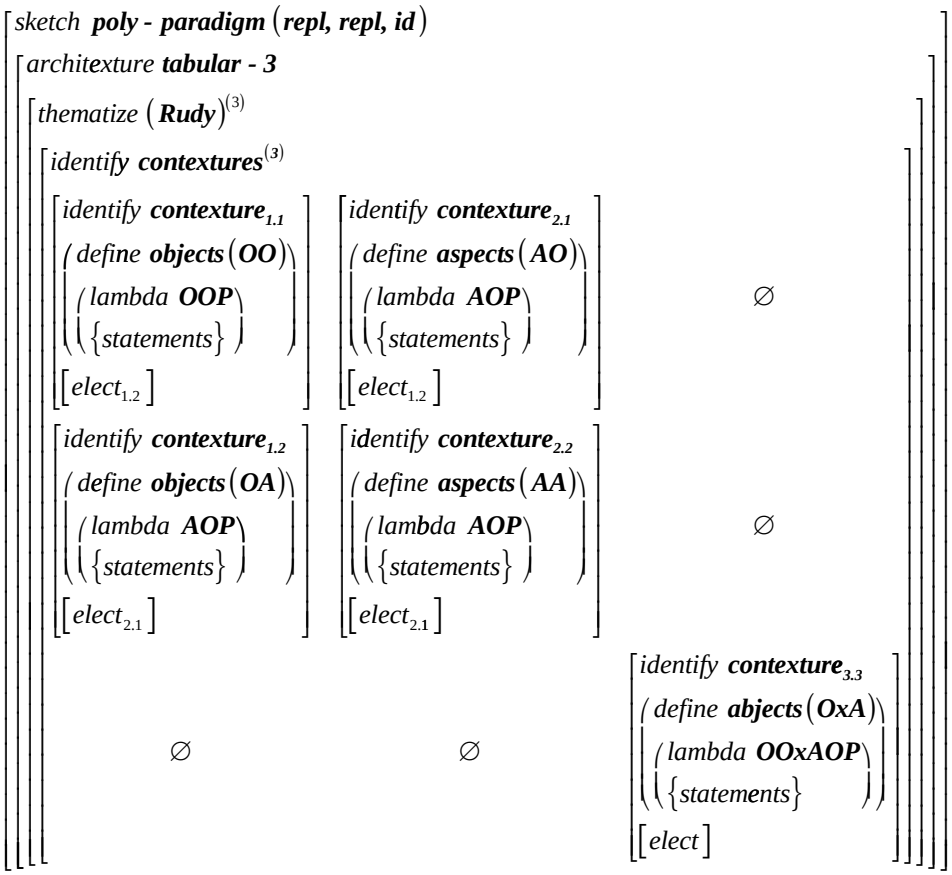
Ruby as OO and as AO and as AxO



Ruby as OO and AO and OOxAOP as Ruby



A fragment of Ruby/Rudy



Conclusion: From Ruby to Rudy

Myth 2: AOP doesn't solve any new problems.
Reality: You're right -- it doesn't!

Nothing, surely has changed for AOP. You can't solve any new problems, you also can't design new problem spaces with AOP, neither with OOP.

But all has changed dramatically for Ruby$^{(m, n)}$. Not only new problem spaces can be designed and unknown problems (re)solved but also new programming paradigms will be invented, designed and developed with the support of this very general framework of programming. Thus, it shouldn't be called Ruby$^{(m, n)}$ anymore, but *Rudy$^{(m, n)}$*. Elsewhere I called it *ConTeXtures*.

Remembering Ruby's Intro

To play the game a step further it should finally be understood that also trinity in the Occidental Middle Age was a great deliberation from Greek dualism, today it is a serious restriction and obstacle. Statement-based programming should evolve to a new deliberated paradigm *calligraphed* in patterns and their thematizations. Thus, from POP, OOP, AOP we should move, at least, to MAOPP (Morphogrammatic Aspect-Oriented Programming Paradigm) to be prepared and to stay competitive to the *Chinese Challenge* (George W. Bush).

Time and Computation

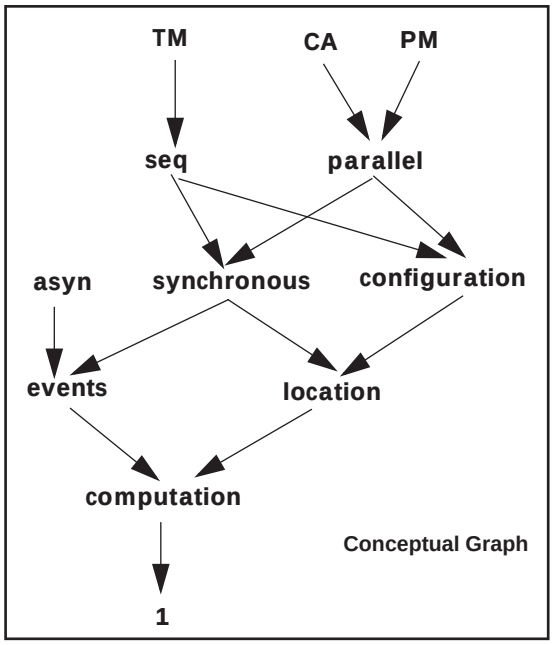
1 Linearity of computational time

Also the notion of time is according to Augustinus *"A well known unknown"*, it has developed in the history of alphabetism and logocentrism to a trivial truth, that time is an absolute and linear ordered sucession of events in the modi "past", "present", "future". That is: past –> presence –> future, short: V –> G –> Z.

Linearity of time goes close together with the basic structures of semiotics, arithmetic and logic, i.e., concatenation, counting and deduction. Time structures are basic for all kind of computation (Levin). Thus, a lot of research has been done to study all kind of aspects of time from philosophical to logical and computational aspects of time.

http://www.vordenker.de/ggphilosophy/die_zeit_vgo.pdf
http://www.ingentaconnect.com/content/imp/chk/2005/00000012/00000003/art00003

General temporal structure of computation



"*Computations* consists of *events* and can be represented as *graphs*, where edges between events reflect various relations. [...] We will study only *synchonous* computations. [...] Nodes and edges will have *attributes* called labels, states, values, colors, parameters, etc. [...] Their nodes have a *time* parameter. It reflects logical steps, not necessarily a precise value of any physical clock. [...] Rigid computations have an other node parameter: *location or* cell. [...] Locations have a *structure* or *proximity* edges between them. [...] Combined with time, it designates the event *uniquely.*" (Levin)

TM: Turing Machine, CA: Cellular Automaton, PM: Pointer Machine.

On the base of this abstract concept of computational space and time, *computational complexity* can be introduced and studied.

Computational time and space

"The greatest depth $D_{A(x)}$ of a causal chain is the number of computation steps. The volume $V_{A(x)}$ is the combined number of active edges during all steps. Time $T_{A(x)}$ is used (depending on context) as either depth or volume, which coincide for sequential models. $S_{A(x)}$ of a synchronous computation is the greatest (over time) size of its configurations."

L. A. Levin, Fundamentals of Computing, http://www.cs.bu.edu/fac/lnd/toc
http://www.thinkartlab.com/pkl/media/SKIZZE-0.9.5-medium.pdf

Contextural computation cannot assume a homogenous time-space medium for its computations. Changes of contextures are prior to continuations of states inside contextures and are creating a new, tabular kind of time-events and time-structures.

2 A time-matrix for complex object-schemes

After the introduction of complex object-schemes with all their different functionalities as objects, aspects, abjects, injects, etc., questions about their temporality are naturally arising. As a first step, characteristics of a time-matrix and contextualizations of time-modi are sketched.

2.1 Temporal structures of cognitive systems

What happens temporally inside a computational system? How have we to design it that it can happen?

To get a complex temporal structure, a system has to offer a corresponding complex computational structure. As a minimal condition, a cognitive system has to be able to distinguish between itself and its environment and to map this distinction into its inner environment. Its behavior has, additionally, to reflect the fact that it itself is recognized as a part of an inner environment of another cognitive system which is able to distinguish between itself and its environment and to map this distinction into its inner environment. A cognitive system has to realize this complex relational structure at once. Thus a complex cognitive system is an interactional and reflectional societal compound system. As such a cognitive system, it is developing a complexity of different temporal behaviors depending on its behavioral structure.

Physical or simple systems (in the sense of Robert Rosen), like computational systems, don't have an environment. They are, from the point of view of an external observer, placed in an environment, but not *having* an environment; neither inner nor outer environments.

Time as temporal succession

Each aspect of a complex entity has its own internal time-structure. The way the aspects or functionalities are mediated is determining the mediation of the time-structures, i.e., the interplay of different time dimensions, measures, rhythms, etc. of the complex entity. Thus, the temporal functioning of a complex object in a computational environment is depending on its mediational structure. This is producing some restrictions in the combination of different temporal dimensions, but avoiding chaotic constellations. Chaotic constellations appear as partial or total break down of mediation. This has to be considered and studied for real-world constellations. Not every mediation succeeds at the time needed.

Time as change of functionality

The change from one functionality (aspect) to another functionality (aspect) of a complex object is defining an intrinsic time-structure. That one functionality can become another one, say an object can become an aspect, and vice versa, defines an intrinsic dynamic of the complex entity. Such a dynamic is intrinsically timing (offering a time structure) the entity and its system. Thus, the more distinctions an entity is realizing, say as object, aspect, abject, project, inject, etc., the more complex, i.e., the more lively its behavior appears. That is, timing as an intrinsic temporal dynamic is a chiastic interplay of the entity's functionalities.

Time as a second-order category

Such time-mechanism, internal and intrinsic, are essential characteristics of the behavior of interactional and reflectional computational systems. They are not accessible by external observations of the behavior of the system.

Thus, time is depending on the point of view and focus of an observer, internal and external. And it is changing relative to the change of the point of view of the observation. Computational time and timing is not the same as physical space-time.

Time in temporal logic

These considerations have to be, at first, strictly separated from enquiries resulting in modal logic based temporal logic (Manna, Pneuelli).

"In temporal logic, the interpretation given to the accessibility relation is that of the passage of time. A state s *is accessible from another state* s' *if through a process in time* s *can change into* s'*."*

The proposed studies I am sketching are dealing with time and timing (Zeitigung) and not with the change of states in time. The common approach in logic and computer science is presuming time as such and is studying what happens to the states of a systems in such a succession of time. Time in this sense is a kind of physical time, which is a property of an external description of the behavior of the system under observation. The notion *time*, then is classified as *linear* or *branching*.

> *"We may distinguish between two notions of time: branching time temporal logic, and linear time temporal logic. In branching time temporal logic, we view time as having a tree-like nature inn which, at each instant, time may split into alternative courses representing different possible futures. In linear time temporal logic, we view each moment as having only one possible future corresponding to a history of the development of the system."* Jonathan S. Ostroff, p. 155

Also formalized, the temporal basics of past/present/future are kept on. A state *s* is before or after a state *s'*, independent of linear or branching time. But depending on a temporal measure. Other topologies of temporal time structures are not denying this basic order. http://plato.stanford.edu/entries/logic-temporal/

2.2 General tabular time-matrix

Computational systems, in the common sense, are similar to physical systems. They they are running in the framework of an observer independent time concept. It is possible, theoretically, to identify, but also to separate, each single state of the system from its successor or predecessor state. In contrast, complex or cognitive computational systems are much more shaped by holistic and emergent properties. Thus, time in such systems is not a linear succession of states or events but a complex chiastic interplay of interacting and reflecting parts.

We can mirror this situation to similar properties of living systems.

According to Langton (1989),

> Linear systems obey the superposition principle since they are decomposable into independently analysable components and composition of understanding of the isolated components leads to full understanding of the system. The principle does not hold for non-linear systems since in this case, primary behaviours of interest are properties of the interactions between components as contrasted with properties of the components themselves; isolating the components necessarily leads to the disappearance of interaction-based properties.

A first and simple step to such an interplay of aspects of time can be considered as a time-matrix of the reflected modi of time, "past", "peresent", "future". This kind of reflection follows the *as-abstraction*.

Classical time modi are in the *is-abstraction*: X is X. That is, a *state* in the past is a state in the *past*. Such a state of the past remains a state of the past; it can not become involved in a future. The is-abstraction is an operation of identification and not of interpretation or thematization. Thus, this notion of time is strongly related to the ontology of states and time is state depending. There is no structural space for time as such.

Reflected time-modi

GG: presence as presence: now (Jetzt)
GV: presence as past: retro (Rückbesinnung)
GZ: presence as future: design (Entwurf)

VV: past as past: past (Gewesene, Vergangene, Vergangenheit)
VG: past as presence: tradition, inheritance (Tradition)
VZ: past as future: interpretation, re-definition (Umdeutung)

ZZ: future as future: future (Künftige, Zukunft)
ZG: future as presence: arrival, event (Ankunft, Heraufkunft)
ZV: future as past: plan, design (Entworfene, Beikunft)

VGZ-time matrix

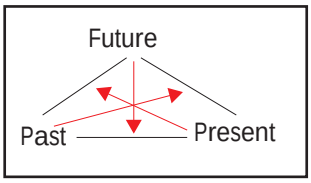
TM	*V*	*G*	*Z*
V	*VV*	*VG*	*VZ*
G	*GV*	*GG*	*GZ*
Z	*ZV*	*ZG*	*ZZ*

The time-matrix is representing the contextuality of the 3 time modi, past, presence and future, in their dependencies. But, as a first step out of the linearity of the classic understanding of time, only a 2-dimensional inter-relation is introduced. The 2-dimensionality is modeled along the as-abstraction: X as Y.

With the time-matrix a tabular notion of time is conceptualized. The classic linearity of time is preserved in the time-matrix as the diagonal structure of the matrix. Connections to possible temporal modi in grammars of natural languages are not yet considered in this proposal.

Contextualized time-modi

Time-modi as view-points to characterize the relationship between time-modi.



As a time-mode can be characterized as it is, past is past, and time-modi can be reflected as being different, past as future, the relationship between different time-modi can be characterized from the point of view of other time-modi, e.g., the past-present relationship from the position of the future, etc. Thus, changes in the relationship between time-modi can be founded by their contextualization, i.e., by its internal or immanent view-points of description. That is, from the point of view of the past, the structural relation between future and present in respect of its relevancy can change. If, from the view-point of past, future was dominant, it can change to an inverse order. And future may be in different constellations relative to the present and to the past. From the past, future in future / present may be relevant. But future in the relation future / past may not be relevant from the view-point of present. Thus, the common linearity of the time-modi is transformed by contextualization. In other words, the Present can be understood as the difference between the Past and the Future, as well the Future as the difference between Past and Present, and the Past as the difference between Future and Past. Thus, time is understood as modi of differences and not as the temporality of events or objects.

Multi-dimensional time-matrix

The contexturalization of the time modi has not to be restricted to a 2-dimensional matrix of the as-abstraction X as Y is Z. That is, past as future, future as present, etc. More complex matrices can be introduced on the basis of more complex as-abstractions. This is also including combinations of time-modi, like the future of (past and present), etc.

The presence of (the presence, the past and the future)
The past of (the presence, the past and the future)
The future of (the presence, the past and the future).

2.3 Classification of temporal events

Without doubt there are other types of classification of temporal events possible than the classic "past-presence-future" approach. Depending on language, notational systems and cultural tradition classifications beyond the Greek model are possible.

The *modi of time* are depending on the complexity of the involved reflectional / interactional systems. There is no time out there without a thematization and interpretation of the world by a living system. Time is an aspect of thematization of the world and not an absolute attribute to the events of the world. An absolute, relativistic or not, understanding of time and space occurs as a reduction of the complex thematization of time. In a world without reflectional agents an objective linear time notion is appropriate, say for chronology and other measurement of non-living systems. Thus, the restriction in this contemplation to triadic-trichotomic structures in semiotics, logic and time modalities, is only an introductory and temporal restriction, without systematic value.

Full time-structure of a computational system

The full time-structure of a computational system, thus, has to realize, at once, its *internal* and its *external* modi of timing. This can be observed and described by a "double" action of observation and inscription, only.

How much it is still a problem to think such a "double science" to understand the simultaneity of system and environment, i.e., the system-environment coupling, is witnessed by the following paragraph:

> However, this description is incomplete since only the topology (structure) of the system-environment coupling has been specified. In order to complete the description it is necessary to define each component and each relation. Furthermore, as Heylighen (1993) states, "no absolute distinction can be made between internal and external, that is, between system and environment. What is`system' for one process is `environment' for another one." (p.3) Hence, the system-environment coupling relations are relativistic irrespective of whether the relativism is functional (ontological) or observational (epistemological).
>
> S.M. Ali, R. Zimmer, Discourse On Emergence: Part I. (Foundations)
> mcs.open.ac.uk/sma78/disc_1.pdf

Who wants an *"abstract distinction"*? From which point of view could it be established? Would it then still be abstract? Whatever *"relativistic"* means, the description, *"What is`system' for one process is `environment' for another one."*, makes the chiastic interplay between system and environment, depending on different positions, clear enough to be modeled and formalized in a polycontextural setting. As a further step of reflection, the interplay of system and environment itself can be considered, and named in this specific context, an abstract distinction. Thus, the abstract distinction is constructed as the process of mediation between system and environment.