ELSEVIER

Information and Software Technology 44 (2002) 617–638

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Molecule-oriented programming in Java

Jan Bergstra[a,b,*]

[a]*Programming, Research Group, University of Amsterdam, The Netherlands*
[b]*Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 Utrecht, CS, The Netherlands*

Received 18 September 2001; revised 24 January 2002; accepted 3 April 2002

Abstract

Molecule-oriented programming is introduced as a programming style carrying some perspective for Java. A sequence of examples is provided. Supporting the development of the molecule-oriented programming style several matters are introduced and developed: profile classes allowing the representation of class protocols as Java classes, the 'empirical semantics' of `null`, a jargon for the description of molecules, some terminology on software life-cycles related to molecule-oriented programming, and the notion of reconstruction semantics (a guiding principle behind the set of case studies).

Keywords: Object; Molecule; Molecule-oriented programming; Focus; Field; Java

1. Introduction

This paper covers a number of issues regarding Java programming. For an introduction to Java and the ideas of its designers we refer to the rapidly growing textbook literature.[1] A very informative recent introduction to Java is Ref. [10]. The relevance of the issues raised relates to programming methodology and style. Explicit work on Java programming style is rare in an enormous stream of pragmatic work. We mention [1] as an example of a text using an interesting style: preconditions and postconditions phrased as comments.

Central to the paper will be the viewpoint that the construction and modification of molecules consisting of objects (playing the role of atoms) and connected by fields (playing the role of bonds) provides a meaningful operational model for Java programming. Here are our objectives, proposals and results, collected in itemized form.

- The reconstruction of elementary mathematics as well as elementary theory of computation and formal logic in Java is an interesting task. Even the description of the natural numbers (NN) poses some challenge. Below a so-called perfect class representation (for NN) will be given. In a perfect class representation of a mathematical domain each object from the domain is represented by exactly one object from the class of representations. A similar definition is provided for the integers (integral numbers, IN).
- Java fails to offer the Cartesian product as a primitive. Neither are sets native to Java. Further there are no generic classes and no higher order functions. By introducing perfect classes, cartesian classes and non-cartesian classes some structure is obtained guiding a path through the complications caused by the need to do without these familiar features. It is open to debate whether the absence of a number of prominent features in Java must be considered a weakness or even a flaw. These matters are illustrated in a non-perfect class representation of the complex integers (CIN).
- This paper contains the printouts of a number of Java class source files. The main expository problem generated by printing the class source texts lies in the disappearance of (abstraction from) the locations in the directory structure and the consequences of these matters on the import and use relations between source files (and class after compilation between class files). In order to compensate for this loss the JCF (Java class family) notation is used. This notation allows one to describe exactly how a number of class sources are placed in different files and directories. These aspects are of course marginal from a general perspective. In practice the whole systematics of naming conventions and 'configuration management in the small' cannot be ignored, however. Many classes make sense only in a very specific

* Address: Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 Utrecht, CS, The Netherlands.
E-mail address: jan.bergstra@phil.uu.nl (J. Bergstra).

[1] The author acknowledges numerous suggestions for stylistic improvement by R. Roël, J. Fokker and P. Rodenburg.

0950-5849/02/$ - see front matter
PII: S0950-5849(02)00065-4

context. The JCF notation allows one to produce a readable printout of such a context. It is evident how to represent JCF's in XML applications when needed.

• Interfaces in Java cannot be used to define abstractions of arbitrary classes, because of a complete neglect of static aspects and because of particular typing conventions. Several remedies against this fact have been developed. For instance the Javadoc tool generates 'protocol' descriptions for classes that are used massively in the textbook literature on Java. What we look for is a notion of an interface that captures all the interface properties of a class and simultaneously admits type checking by the compiler (just as interfaces do). Abstract classes cannot be used for this purpose either due to similar limitations.

Below so-called profile classes are introduced. For most programs in the paper appropriate profile classes are presented. As we intend to work with the Java system in the form available to us, no modifications of Java supporting profile classes have been contemplated. The profile classes below show some unfortunate details forced upon the author by limitations (or conventions) of Java. It is vital to have the names of profile classes and Java source classes identical. As a consequence the standard heuristic that class texts ought to be placed in text files with identical names is violated. (We found it very practical to use the option to have class source texts in files with different names in many occasions!)

• The 'object' `null` plays a role in many programs. However, it turns out to be remarkably difficult to develop a coherent view on the status of this 'thing'. Right from the start the idea that an apriori theory can (or should) be provided explaining a prospective programmer what to expect from `null` is rejected. Developing such theories is a subject of its own, the pitfalls of which are known to everyone who followed the development by trial and error of proof systems for the equational logic for partial algebras. The method for semantic clarification proposed here at least is termed 'empirical semantics'. Empirical semantics may be helpful in some cases. In empirical semantics one tries to obtain some coherent picture of the role that a particular program construction (e.g. `null`) can play by experimenting with an informative series of example programs. Having studied these examples the reader has some opinion on what to expect, and the designers of formal theories have crucial information of what direction to take (if explaining how things actually are in Java still is their target!).

A major reason to abstain from formalized modeling is that the formal models are more likely to explain 'how things should have been' than how things actually are (in some cases at least). As a case study of empirical semantics the properties of `null` are investigated in some detail. The author considers the picture emerging quite unsatisfactory, but that subjective judgment is irrelevant for Java programming of course.

• The position is taken that an object in Java is a point (node) in a directed labeled graph. Configurations of connected objects arising during a single computation will be referred to as molecules. Molecules are identified as the main structuring concept needed to obtain a model of the dynamics of a Java process (=the execution of a Java program).

There is no obvious representation of many subject domains in the world of Java objects and molecules. Writing programs designed as to generate molecules with a well-understood structure and dynamics is termed molecule-oriented programming. Molecule-oriented programming is neither directed towards the realization of a particular functionality nor is it aiming at some particular program structure or format. The objective is to model concepts in terms of the dynamics of a world of molecules.

• Java allows the programmer to design a dynamic, directed and labeled graph (hereafter often referred to as a molecule). The objects are nodes, while the fields are labeled arrows from node to node. Instance field selection takes place by following an arrow from one object to a second one. At any time, classes are subsets of the universe of objects. Objects cannot leave a class.

A chemical metaphor is proposed to exploit the (potential) similarity of programmer defined classes with classes of chemical atoms as classified by the periodical system.

• Reconstructing known structures and theories can be considered an activity in semantics. This activity is termed reconstruction semantics. Reconstruction semantics (using Java) aims at a complete explanation of a topic within Java in such a way that a theoretical preliminary foundation in terms of other theory is redundant. Below we will produce a reconstruction semantics in Java of natural numbers in unary notation (NNj), the integers (INj), the complex integers (CINj).

• Conceptual programming is coined as a phrase denoting a kind of programming geared towards examination of the conceptual issues arising from the representation in Java of structures and systems emerging from different topics.

The examples demonstrate the use of the molecule-oriented programming technique in a conceptual programming effort aimed at giving descriptions of numbers and lists.

2. Program theory

Program theory addresses amongst other questions the following issues.

- What is a program? What is a program notation? What is a program text, a compiled program and what is program behavior?
- What are the basic program construction mechanisms (programming language features); how should program notations be classified according to the features offered?

- What is a program specification; when is it the case that a program meets a specification; how can this be tested, proven or refuted? How are tests to be assessed, proofs to be checked and refutations to be validated?
- What is programming; how to design programs given specifications to be satisfied beforehand?
- What can be programmed; where is the programmer confronted with the limits of computability, either in principle, or (just as important) in practice because of current technical limitations?

2.1. Program algebra

There are many Program Theories. As a classical example [5] can be mentioned. The author uses program algebra (in the style of PGA, [4]) as a vehicle for program theory, the pro's and con's of that choice being of no importance to the subsequent considerations, however. For a survey and critique of program theories see Refs. [7]. Program algebra provides a line of thought that provides a basis for the assertion that Java program texts indeed represent computer programs. Other program theories provide different, but equally valid, explanations for that assertion.

Program algebra declares that a program is a non-empty sequence of so-called basic instructions. Infinite sequences of instructions can be obtained using repetition of finite sequences. In the simplest case, sufficient for computer programming, the infinite instruction sequences are periodic.

A text is a program text if it represents an instruction sequence. The transformation from a program text into an instruction sequence is called a projection. Projections are in general defined uniformly for all programs in a program notation rather than just for individual programs. The semantics of a program text is found by attaching a behavior to its corresponding program object found after projection. Indeed the meaning of a program text is found via its program object. The projection leaves the meaning invariant, or put differently: the projection defines the meaning. For a text to qualify as a program text a projection must be at hand. Only in combination with the procedure of the projection can it be maintained that the text in fact represents a program.

2.1.1. Process algebra

The meaning of a program object in turn is obtained by means of behavior extraction. This is a transformation turning the program object into a behavior (or process). This transformation can take many forms and it depends significantly on the choice of a theory of behavior. The author's favorite choice of a theory of behavior is process algebra (in the style of ACP, [2]).

The behavior extracted from the projection of a program text contains information that is likely to be refutable by empirical studies of running programs. Behavior contains information of the effect (German: 'Wirkung') of a program, whereas the projection contains information about the mechanisms playing a role in generating the behavior (in Dutch: 'werking'). The projection determines the 'werking' of a program, thereby explaining its 'Wirkung'.

2.2. Java program texts and Java programs

Given a fixed Java version and compiler the argument that a Java text is in fact a program text runs as follows. A Java text is a text accepted by the compiler as a valid program (errors are not allowed, warnings may be ignored). The compiler transforms accepted program texts to a so-called byte code. This byte code can be disassembled into a program in a very simple program notation resembling an assembly language. By unfolding the loops in that program an instruction sequence is obtained, qualifying to serve as a projection of the program. A this point no more is said than that it essentially is the compiler (and its capacity to serve as the major part of a projection) that justifies the qualification of a Java text as a program text. Presenting the details of an actual Java projection is quite another story, well out of scope for this paper.

The phrase 'Java program' is just an abbreviation for 'Java program text'. For a text to be a Java program it is essential that a compiler is known. Different compilers may transform the same text into different program objects (instruction sequences), leading to different behaviors in turn. The text exists independently of a compiler (projection) of course, the program text does not, however.

3. Java programming

Programming is understood as program text construction. Many phases and cycles can be distinguished in program text construction, mostly linked to general objectives. At this point it is relevant to identify the simplest possible conception of program text construction, irrespective of its methodological or practical merits. Remarkably the literature on the design process is more systematically developed than the literature on programming seems to be. We mention Ref. [9] for a very thorough definition of the design process.

3.1. Program text construction cycles

Three cycles are distinguished: the syntactic program text construction cycle (PTCCsyn), the empirical program text construction cycle (PTCCemp), and the rational program text construction cycle (PTCCrat).

Much can be said about quality control in relation to these cycles. The foundations of such considerations are part of program theory, except in the case of the rational

cycle. Leaving program theory out of consideration, no substantial progress on quality control can be achieved.

3.1.1. The syntactic program text construction cycle

In PTCCsyn a text is produced and submitted to a syntax and type checker. Usually the syntax check for a program notation is incorporated in the tasks of a called compiler, for that reason below 'compiler' will be used also if 'syntax checker' is meant. In reaction to bug reports the text is transformed until it passes the syntactic requirements posed by the compiler. Only at this point is the text a program text. Only program texts are called programs (when using abbreviated language). Before approval by the compiler one may speak of a candidate program text or a candidate program.

3.1.2. The empirical program text construction cycle

The empirical program text construction cycle (PTCCemp) involves the second stage of compilation: code generation. Code generation transforms a program text to 'executable form'. In the case of Java this execution is performed by a so-called virtual machine, itself a program in turn. The generated code is then run on various inputs. Again feedback is obtained, now on program behavior, and modifications are made until the observed behavior corresponds with the program author's intentions to a sufficient degree. It should be noticed that PTCCemp can be performed in the absence of any apriori, formal or informal specifications on the behavior of a program.

The process of writing is completed when the author is satisfied with his/her product. Criteria for satisfaction may vary significantly in different circumstances.

3.1.3. The rational program text construction cycle

The rational program text construction cycle PTCCrat involves quite another feedback mechanism: the program writer reads the text and understands it as an intelligible statement about a topic of value. The program text is rewritten until, as a text readable for human readers, it conveys a story approved by its author. Of course other readers may play a role in this cycle, and subcycles of type PTCCsyn must be expected.

4. Why Java

Java is taken as a point of departure for this paper. The abundance of its use provides ample justification for taking an interest in the minute details of Java programming and its methodology. There will be no emphasis on contemplating extensions of or alternatives for Java. Java is considered 'part of nature' and is investigated as such. Information on 'what the Java designers had in mind' is not taken into account. The emerging traditions of Java programming are equally disregarded, thus allowing an unbiased focus on the Java program notation itself.

This text is not about the mathematics of programming, neither does it contain an attempt to formulate a general program theory. Still the topic is focused on principles of programming in relation to a particular (kind of) program notation. Writing such papers will always require the selection of at least one program notation and supporting compiler. Moreover, for a specific prospective program text, the approval by the syntax checker (included in the compiler) for the chosen notation, of that text should be accepted as a sufficient criterion for being a program text. We notice that Java is quite interesting in its own right: besides a remarkable market penetration (in Academia at least), it provides objects, classes and threads in a more accessible form than any previous program notation has done so far.

From the viewpoint of software technology the ambitions with Java are just as significant: platform independence, security, code transportability and automated memory management are approached in a principled and integrated fashion.

4.1. Instrumental use of Java

People interested in Java programming may either consider Java a toolkit meant for reaching other objectives, or a theme and focus of study and research in itself. Only from the perspective of 'pure informatics' is Java a plausible option as a theme in its own right. We will not take that position and concentrate on forms of instrumental use of Java instead.

Suppose one considers Java an instrument for reaching some pragmatic goals. Assuming a person unaware of program theory one may question which goals are within reach for this person using Java at all. The following goals can be rejected at once:

- The production of safety critical software. This work requires awareness of systematic quality control at the level of program behavior. It is a common opinion that this task is the hardest challenge in programming of all. Without a firm footing in a solid theory of programs no reliable results can be expected. Amateur contributions are pointless just as there is no need for amateurs in pharmaceutical production or in the medical profession.
- Data base applications. These are to be considered safety critical in many cases, and so are many programs for business control. The difficulties are like those with safety critical software.
- The production of high performance information processing tools. This activity requires an up-to-date awareness of algorithms. Again this requires a solid exposure to a theory of programs.

4.2. Conceptual programming: instrumental use without a program theory

Conceptual programming consists of program construction with the goal to understand or explain the subject about which a program is written. Conceptual programs are texts meant for the human reader. The primary production cycle for conceptual programs is PTCCrat (see Section 3.1.3). The criterion for a program to pass as a successful program is its capacity of conveying complex information to its readers as a text. The more high-level program languages are the better suited for conceptual programming. The design of COBOL has been influenced by the ambition to enhance conceptual programming, the language giving rise to readable texts. Conceptual programming is accessible for programmers not aware of any theory of programs. The meaning conveyed by a program is constructed in the mind of a reader, that construction falling outside the scope of any theory of programs developed so far.

4.2.1. Conceptual programming in Java

Conceptual programming in Java is a plausible goal for a person intending to program from an instrumental perspective. The production of conceptual programs is a clear-cut objective, the lack of awareness of program theory being even a potential advantage. Conceptual programming also refers to a programming style putting readability and comprehensibility at top priority at the cost of performance in case the program is executable in a meaningful way at all.

It is a goal of this paper to put forward conceptual programming as a plausible objective thus turning an instrumental view plausible (for a wide range of program authors), while not allowing the deep problems concerning software quality control to dominate other ambitions of program production.

Secondly the paper aims at demonstrating the potential of conceptual programming in Java in some interesting but limited cases.

A very remarkable specimen of conceptual programming in Java is found in Ref. [6]. This book explains a number of complex design patterns in terms of Java programs. It is obvious throughout the book that execution of these programs is of secondary importance.

4.2.2. Conceptual programs and Java presented theories

The following assumptions support conceptual programming in Java.

- A theory can take the form of a family of Java programs (or even a family of families of Java programs) together with an explanation of the rationale of these programs. (Below a theory of natural numbers, integers and complex integers, as well as a theory of lists, is presented in that form.) Such a theory is called a Java presented theory (JPT).
- A formal semantics of Java is not a fundamental prerequisite for the existence and usefulness of Java presented theories. The JDK or any other implementation of Java provides so much semantic information that a Java presented theory may well be clearer than a purely informal theory phrased in natural language. Even a theory containing formalized parts may show considerable unclarities and ambiguities at the borders of the formalization.

Needless to say in some cases the lack of clarity of the semantics of Java may adversely affect the overall comprehensibility of a JPT. Such unclarities will be an invitation for semantic investigation concerning specific fragments of Java.

Nevertheless, a JPT can serve as a description ‘ab initio’ of a subject. The semantic questions concerning Java itself being considered a problem to be dealt with pragmatically in very much the same way as the semantic problems of natural language are dealt with in ordinary theories.

- Java is sufficiently expressive for the development of JPT’s. Disadvantages of Java in comparison with other imperative languages concern efficiency of execution much more than brevity (or clarity) of expression. Java is far more expressive than any logical formalism or process algebra known to the author. As a consequence it is viable to use Java as the main content carrier in a JPT.

5. Molecules and the field-focus distinction

Object orientation lacks a completely fixed jargon. Therefore it is hard to avoid the introduction of specialized jargon if the subject is to be dealt with in great (informal) precision. Here is a proposal for such jargon.

Molecules are coherent constellations of objects (or rather items, see below). The only possible connection (binding) inside a molecule consists of an object (item) being contained in a field of another object. Fields always belong to an object. A field is also thought of as a named arrow from one object to another object or item. The field belongs to its source and is said to contain its target. It may be useful to have an arrow pointing into a molecule from outside. Such an arrow will be termed a focus. A focus never belongs to a field.

5.1. A classification of fields

An item is an object, a value or an external item. A field is a named arrow from one object to an item. The field is said to belong to the object it leaves and to contain the item that it points to. Three kinds of field are distinguished, according to the kind of item contained by the field:

Object field An object field is vacant or it contains a Java style object.
value field A value field points to a value container; different value fields can never point to the same value container. Each value container has its own ‘space’. Value containers contain values. A value field is said to

contain the value contained by the value container it points to.

external field An external field is an arrow from an object to the description of an entity (an external item) that might survive termination of the program. (The field itself, together with the object it belongs to, will disappear when the program terminates.)

5.1.1. A classification of foci

A focus is an arrow from outside (the molecule) to an item. Typically references on a stack and local program variables can be viewed as a focus. The same classification in three categories valid for fields is meaningful for foci as well.

5.2. Comments on fields and foci, independent of Java

Here are a number of remarks and observations meant to get a sharper picture of the various categories at stake.

(1) Every program builds its own universe of objects during execution. All objects are constructed in a sequential fashion. One may think of an object as being uniquely tagged by its time of construction, or its rank number in the temporal succession of object constructions.

(2) If a program terminates all objects and value containers that have been constructed during its execution disappear.

(3) The run of a program starts from a root object. That root object is in practice a so-called thread provided by the system. During the run of a program there is a so-called program stack. This is a collection of names together with their meanings. Such a name will be called a focus hereafter. The names have been introduced during the computation and their meaning is an arrow pointing to an object. Accessible objects are those objects to which a focus points as well as all objects reachable from any accessible object by selecting one of its fields.

(4) An item is either an internal item or an external item. An internal item exists during the course of a computation and ceases to exist thereafter. External items may have a life extending beyond the life-time of a program using a name pointing to the item. Typical examples of external items include: a named file, a (numbered) socket, a URL. An internal item is either an object or a value container. During a computation an existing value container will at any time contain some value.

(5) A value usually has an external meaning in the setting of other theories, computing systems and so on. It is a design decision common to many programming languages to have only values that can be understood as representations of 'generally known quantities'. It is a consequence of the von Neumann machine model, however, that the distinction between small and (potentially) big has to be made. Having a correspondence between this distinction and the previous one, regarding internal items versus external items, is by no means a necessary consequence of the von Neumann machine model.

(6) An object may be 'small', 'medium' or 'big', a value (placed in a value container) is always small and so is its container. From the viewpoint of a computation an external item has an arbitrary size.

Different (object) fields may contain the same object, different external fields may contain the same external item descriptor, similarly different object foci may contain the same object and different external foci may contain the same external item descriptor.

(7) An object field or an object focus may be thought of as an arrow capable of pointing to an arbitrary object. Because the object may be 'big' it can be useful to have different arrows pointing towards the same object. In contrast the values placed in a value container are small, for that reason it is never useful to have different arrows pointing towards the same value container because the arrow by itself is more costly than a value container with value together with a unique pointer pointing to it. Such unique pointers are value fields or value foci. For external items the possibility to have multiple references is almost a logical necessity. In all cases external items may be considered 'big' making it economically plausible to treat them as objects rather than values.

(8) During the run of a program an object or a value container may in fact disappear. The disappearance of an object is not the result of an explicit program instruction, however. It is not possible to find evidence for the removal of the object by means of instructions in the Java program. The removal of objects (and value containers) is taken care of by 'the system'. The system can remove (garbage collection) objects that have become inaccessible (garbage status). Each system can only contain a bounded number of objects (and value containers) at the same time. Removing garbage (garbage collection) is a form of metabolism. It frees memory needed for the construction of new objects. We will discuss molecule-oriented computing as if it takes place in a processor with infinite memory space.

(9) In a discussion of objects various objects get names. This is the case in programs with objects as well. Suppose p and q are two object names. Now object equality, $p == q$, expresses that p and q refer to the same object. This is the strongest possible form of identity, objects p and q cannot conceivably be more equal than $p == q$.

(10) It is reasonable to view the expanding universe of objects for a program during execution as a growing collection, the fields denoting names of arrows from object to object. The effects of garbage collection cannot influence the functionality of a program. The only benefit, and effect, of garbage collection relates to performance.

(11) Complex mathematical structures are often represented as subgraphs of the current domain of a class. A particular structure is then identified with a particular node in the graph (i.e. an object) from which all other elements of

the subgraph can be found by following the arrows (selecting fields).

Elaborating on this point an important problem regarding class naming emerges. Because the 'real structures' a programmer intends to describe (e.g. a search tree) are subgraphs of the domain, individual objects are graph nodes rather than these 'real structures'. A class name TREE can for instance be a suggestive name for a class 'top node of tree'. The entire 'real structure' exists only in an implicit form. Now 'top of tree node' is not a very friendly name, but 'tree' is hardly justified. This constitutes some sort of a dilemma. One may say that the very concepts of mereology: structure, component and part, are absent from object-oriented thinking.

5.3. Molecules in more detail

Here is a listing of definitions and terminology regarding molecules.

1. A molecule is a coherent collection of objects and value containers, closed under selection of value fields, object fields and external item descriptor fields, all reachable from a single object in focus.
2. Various foci may simultaneously point to different objects in a molecule. Like objects, molecules exist at run-time only. Like objects molecules cannot leave the processor on which they have been constructed. (Though distributed molecules are a meaningful concept, unlike distributed objects.)
3. Each (object) focus determines a molecule, the union of all molecules thus obtained is the collection of reachable objects. All other objects have garbage status.
4. The principal mental image of the state of a machine during a computation is that of a collection of molecules each determined by its root focus. The various foci reside on the so-called stack, the molecules reside in the so-called heap.
5. Given a molecule and an object P in the molecule, a submolecule is determined as the collection of objects, value containers and external item descriptors, reachable by (repeated) field selection, from the object P.
6. The size of a molecule can be roughly defined as the sum of the size of its constituent parts. Object sizes in turn, are determined by their class definitions, value container sizes determined by underlying value type definitions, external item descriptor sizes determined by the (maximal) size of such descriptors in the intended external context (e.g. sockets are 'small' ints).
7. In theory molecules may grow indefinitely during a computation. In practice the sum of the sizes of all molecules is bounded by the physical limits determined by the amount of processor memory. Each object, value container and each external field descriptor container takes its own amount of memory (again determined by its size).
8. The molecule-oriented picture is quite abstract indeed. In reality, the garbage collector will be systematically busy to remove entire molecules for which none of the available foci serves as a root any more. In order to make efficient use of memory, remaining molecules will be repositioned in memory. This process is called defragmentation. Garbage collection and defragmentation should be considered tasks outside programmer control and outside programmer responsibility (in principle). (In practice the programmer of safety critical Java software should unfortunately be suggested to take extremely careful notice of these matters, however!)

5.3.1. Molecule types

If one intends to understand a particular Java program, or its design, it may be very helpful to think in terms of molecule forms (types, kinds) rather than in terms of plain molecules. A list, or a tree or a two-dimensional grid can be imagined as different molecule forms. Having no syntax for molecules, Java lacks syntactical support for the description of molecule forms altogether. Such forms may be appropriately described by means of graph grammars for instance.

5.3.2. A chemical metaphor

Chemical elements can be compared to Java classes. Individual atoms can be compared to particular objects. Atoms belonging to the same element is like objects having the same class. Different isotopes are like different subclasses of the same abstract class. A chemical bond is like a field. By selecting that field another atom can be found. Elements are characterized by number and type of these bonds. An element may be bound to itself. Each element in the periodical system can be mapped on a class. Chemical bonds being bi-directional every bond is modeled as a pair of fields in opposite direction. Chemical bonds have additional spatial information absent from Java.

Chemical substances, most notably the polymers, may be considered molecule types. The chemical methaphor is to view (as a model) a molecule type as a chemical substance and a particular molecule as an instance of that substance. The computational process involves a chain of reactions between molecules. These reactions take place at so-called threads. Processing involves mainly the modification of values in value containers, the modification of objects contained in fields, the creation of new objects and value containers, and the interaction with external items (addressed via external item descriptors). Exclusively via the interaction with external items the results of processing become available for the external world. The molecules are merely a means to an end, where the interaction with the external items constitutes an underlying objective.

5.4. Molecule-oriented programming

In molecule-oriented programming the programmer intends to exploit the chemical metaphor.

Each structure is represented by means of its own kind of molecules and operations take the form of reactions between such molecules.

A Java programmer has the liberty to define his/her own 'periodical' system, each element taking the form of a class. Computation is the chemistry governed by this new periodical system.

5.4.1. Molecule-oriented modeling

Molecules can be used to represent various aspects of relevance to computing. Data molecules represent concrete instantiations (states) of data structures and data objects.

In general molecule-oriented programming must be preceded by molecule-oriented modeling. The major aspects of a subject area must be modeled as molecules of a tailored form. This leads to modeling and design in terms of molecules. Java introduces some drastic limitations to the modeling process: objects have only a bounded numbers of outgoing arrows, and that number is always fixed at the time of creation. Further the single inheritance paradigm is a significant handicap. Quite simplistic forms of multiple inheritances would prove very helpful indeed for molecule-oriented modeling, design and programming.

Clearly molecule-oriented modeling is mainly meaningful if a representation of important content matter in terms of graphs and graph transformations is feasible.

6. Objects and classes in Java

Objects can serve as a basis for conceptual programming in Java. When being executed on a machine, Java programs give rise to the progressive construction of a number of objects. It is attractive to assume that objects never disappear, until program termination or crash, thus leaving garbage collection outside the semantic perspective.

The intuition of objects as being constructed in time challenges directly the mathematical and set-theoretic assumption of eternal existence. The conceptual programmer is free to reconstruct parts of mathematics from the Java perspective and to see what this brings. The conceptual programmer has succeeded if he/she likes what emerges. The following intuitions may serve the conceptual programmer.

- At any moment a class is a collection of objects. A class description provides the instruments (methods) for the construction and use of the objects in a class. In addition it provides names for so-called fields. A field is a link to another object (or a value).
- All objects are contained in the class `Object`. This class is a superclass of all other classes. Objects are constructed by means of a method named identical to the class prefixed with the keyword `new`.
- The subclasses of a class have extended descriptions, because additional properties or aspects appear. Class description extensions are complementary to subclass extensions. The two uses of 'extension' differ in meaning: an extended description is a larger text, a subclass extension is the collection of objects of a class that belong to the subclass. The principle of extensionality reads as follows for sets: subsets having the same extension are equal. A corresponding principle fails to hold for classes. Indeed two (sub) classes may be empty at some stage. Later on an object may be created for one of them. The absence of objects is no justification for the identification of classes in any stage, however.
- For brevity's sake class descriptions are simply called classes in Java, class description extensions becoming class extensions (i.e. subclasses) as a consequence.

The local variables of a method body, the 'constant' `this`, the static variables of a class and the parameters of a method are all categorized as 'focus'. Only the instance fields of objects are classified as 'field'. Usually fields are declared in a class definition. In the case of anonymous classes a field may be introduced for a single object in the absence of a general class definition, however.

This convention changes if a class is itself viewed as an object as well. Then it becomes more plausible to consider a static field a field (in the sense of(FFD) rather than a focus.

- Each object will always (i.e. during its entire life time) have the same number of outgoing arrows. The same collection of fields will be connected to an object during its entire life-time. Because the object is created as a member of a class, it takes from that class the number of its fields as well as the name of each field and class of objects to which the field will refer.
- The size of an object (in Java) is completely fixed during the definition of its class. Whenever an object is constructed it has a type (a class or an anonymous class). That type determines the number of object fields, the number of value fields and the number of external fields belonging to all objects of the type. The sum of these numbers provides a rough indication of the size of the object. It is irrelevant whether these fields are classified as private or public. That classification relates to their use but not to their existence.
- Different subclasses of a class cannot intersect (unless one of the two is a subclass of the other one). Therefore intersection is not available as a class definition mechanism. The degenerate object `null` is contained in each class, however. It is easy to imagine the description of a class extending two other classes at the same time. This is class intersection or multiple inheritances. For instance: an employee of the University of Amsterdam living in Amsterdam is a member of UvAEMPLOYEE and of AMSTERDAMCITIZEN at the same time. Java will not allow this kind of class description. It has been ruled out (fortunately?) because of implementation problems. In restricted cases where multiple inheritances induce no difficulties its absence of the language is unfortunate, however. As an example consider two classes having only static methods and fields and no name space overlap of any

kind. Taking the 'union' of these classes is very natural and utterly unproblematic. In logic this corresponds syntactically to taking the union of two signatures and semantically to the construction of a common expansion of two algebras.

• Java disallows the use of classes parameterized by other classes. However obvious the intuition of SEQUENCESOF(X) may be, it is not provided in Java. Instead for each particular class (say NN) the class SEQUENCESOF(NN) can be introduced. Unfortunately this leads to a significant duplication of code. This may be considered a real deficiency and it may even constitute Java's single most important flaw.

• Java provides a limited collection of value types. Only elements of those types can occur as the contents of a value field. The actual collection of value types for Java is an ad hoc design decision. There is no theory explaining this part of the design. Array, vectors and strings are in between of values and objects. The molecule-oriented programming jargon has no counterparts for these concepts. None of them is needed for conceptual programming, however.

6.1. The mathematical world of ideal objects

One of the biggest disadvantages of modern computing jargon is the tendency to overload all existing terminology, the term object being no exception. This requires explicit measures for fighting confusion. For instance it is still assumed that a reader has the intuition in mind of the infinite collection of all natural numbers, starting with zero and each made by repeated application of a successor function. This view of the natural numbers leads to the so-called unary notation for natural numbers: 0, $S(0)$, $S(S(0)$,… Often all but the brackets surrounding 0 are omitted: 0, $S(0)$, $SS(0)$,…

This infinite collection contains ideal objects to which no temporal, spatial or other physical attributes need to be attached (or can be attached). The number 7 exists in this ideal world. When working in Java the objects get 'real'. This means that in some sense the objects constructed during the execution of a Java Program have a true physical existence. Even if very little is known about this physical form still it is a reasonable assumption. Java objects (in a program under execution) exist in a physical world inside a computer. Of course the conceptual programmer thinks in terms of a mental model of the execution, thereby giving (hypothetical) Java objects an ideal status as well.

6.2. Java objects and math objects

Math objects are the ideal objects of mathematics, whatever philosophy of mathematics the reader adheres to. Java objects are the physical objects generated by a Java program text during execution. Also in the discussion of a Java program as if it were being executed the term 'Java object' will be used. This is an abuse of language as these objects are just as ideal as any of the Math objects and exist inside a model of the Java execution (however abstract) rather than in any machine in a physical sense.

As a rule collections (sets) of Math objects will be denoted NNm (Math objects for the Natural Numbers) or SEQVm (Math objects for sequences of Math Objects from V). The tag m indicates that Math objects are meant. Java objects will be tagged with a j. NNj is a class of objects in Java for representing natural numbers.

6.2.1. The operator `mathObj()`

When discussing and performing conceptual programming in Java the additional clarity of object existence will be exploited. There is no substitute for the mathematical idealizations, however. Set theory will be used as a carrier for the mathematical intuitions. The set NNm contains these idealizations in the case of natural numbers. The operator `mathObj()` will be used to transform Java objects into their mathematical counterparts if these exist.

6.2.2. Preliminary math objects

Before starting to perform any conceptual programming at all it is useful to determine which sets of ideal objects may come into play. The operator `mathObj()` is used to map Java objects into mathematical objects. Unavoidably `mathObj()` is a construction in the (human) mind.

It is not at all necessary to spell out in detail the mathematical definitions of all objects that might be of use. It is very relevant, however, to know in advance some mathematical sets that will be of use. In all cases the distinction between Math objects and Java objects must be easy to make, avoiding philosophical confusion right from the start.

The value of `mathObj()` need not be defined for all possible objects. In many cases, however, a clear abstraction exists transforming a Java object into a Math object.

6.2.3. Java objects can represent math objects

A most natural intuition is to consider a Java object q to be a representation of `mathObj(`p`)` (provided `mathObj(`p`)` has been given a meaning.)

A class Cj can be considered a representation of a set Sm if all Java objects of class Cj represent Math objects in Sm and if for every Math object in Sm there is at least one Java object representing it.

The representation of Sm by Cj is called perfect if different Java objects always represent different Math objects. Because math objects are made from sets and Java objects are essentially made from texts it is not at all obvious that all sets Sm can be represented in an intrinsic way by means of a suitable Java class Cj. In general this seems not to be the case. The construction principles for sets (Math objects) and Java objects are quite disparate in fact and connections are not easy to find. There is a degree of freedom in the theory of Math objects.

7. JCF notation and Java syntax

All programs are collected in JCFs (Java Class Families). The use of JCFs has been advocated in [3]. The JCF notation is an instance of the folder hierarchy notation (FHN) which has been introduced in Ref. [3] as well.) The details of FHN are spelled out in an appendix to this text. FHN allows one to provide a precise textual representation of the files in a directory as well as a gradual build-up of an FHN description by means of the union operator. On text files union behaves as concatenation, whereas on folders (directories) it acts like a set theoretic union.

The use of FHN and JCF notation in the examples below speaks for itself. The complications of JCF-notation are with folders and packages and these will not be used in the examples below. A JCF is a set of class description files. Each class description file has a name (e.g. `myclass.-java`) and content. The content is an ASCII text containing one or more Java classes.

With JCF-notation it is possible to have many different class descriptions with the same name in one document without causing confusion.

The first JCF contains a class `s` from which all other programs are activated as well as some abbreviations for console output actions. JCFsp = `file:s.java(`

```
class s {
  public static void
    main(String x[ ]){
      (new c()).m();}
  }
) ∪ file:co.java(
public class co {
  static public void p(boolean x) {
    System.out.println(x);}
  static public void p(char x) {
    System.out.println(x);}
  static public void p(int x) {
    System.out.println(x);}
  }
)
```

Below there will be many extensions of JCFsp to larger JCF's comprising a class `c` with a static method `m()`.

7.1. Java syntax: assignment and identity test

The phantom object `null` denotes the absence of an object. It is probably the most ubiquitous object in Java. Whenever an object is created all its object fields point to the `null` object. (If the field is a value field the compiler chooses an initial value itself.) A method cannot have `null` as its target (otherwise an exception emerges). There is only one `null` because `null == null`.

It may be noticed that always `X == X` and `X.a == X.a` hold, whereas `X.a() == X.a()` may fail in case the evaluation of the method `a()` on the target denoted with `X` modifies the state, the method being invoked twice before evaluation of the test.

In a test `X == Y` it is checked whether the focus `X` and the focus `Y` contain the same object at the time of execution of the test, provided both foci contain an object (counting `null` as an object). Otherwise both fields must contain a value of the same type. Then value equality is checked and returned. In all other cases an exception will be raised.

As an imperative program notation Java's most important primitive instruction is the assignment. In an assignment `X=Y` the content of the object focus `X` is replaced by that of `Y`. Immediately after the assignment the two foci contain to the same object and a test `X== Y` will succeed. (If `X` and `Y` are value foci, the type of value containers they point to must be comparable. If not an exception is raised, if so the value contained in `X` is replaced by the value contained in `Y`.)

The assignment `X.a = Y` indicates that the field named `a` from the object in focus `X` will now contain the object in focus `Y`. For this instruction to be executed properly it is required that `X` contains (at the time of execution) an object of some class `A` which has a member field `a` for objects of some class `B` to which `Y` is a focus. Just after the processing of the assignment instruction the following postcondition is valid: `X.a == Y`. Instead of a reference `Y` an assignment can use an expression that produces an (unnamed) reference to an object of class `B`. The reference `X` may not refer to `null` before the execution of the assignment (otherwise an exception will be generated).

In an assignment `X = Y.a` the content of the object focus `X` is replaced by a reference to the object `Y.a` (i.e. the object contained in the field `a` of the object contained in focus `Y`.)

In the assignment `X.a.b = Y.c.d` the following happens: the field `b` of the object contained in the field `a` of `X` is made to contain the object obtained by first selecting field `c` of `Y` and then selecting field `d`.

7.2. Empirical semantics of `null`

The role of the object `null` requires some attention. In the following program several properties of this object have been collected by means of an example. This style of collecting semantic information is called empirical semantics in Ref. [3]. This collection of information regarding `null` in Java may also serve as an indication of the usefulness of empirical semantics. So we take Java as it runs on the machine as a given fact of life (JDK semantics).

7.2.1. Typed versions of `null`

JCFnullex = JCFsp ∪ `file:c.java(`

```
class A1 { }class A2 { }
  void f(A1 x) {co. p(true);}
  void f(A2 x) {co. p(true);}
  }
```

```
class c{
  static void m() {
    BB B = new BB();
    A1 x = null;
    B.f(y); //prints true
    A2 y = null;
    B.f(y); //prints false
    co.p((Object) x == x); //prints true
    co.p((Object) x == y); //prints true
    co.p(((Object) x) == ((Object) y)); //prints true
    /* co.p (x == y);
      generates compile time error: A1 and A2 are incompatible
      types. Apparently == is not transitive!
    */
    co.p ((object)x == null); //prints true
    co.p(x == (A1) null);
    Object o1 = (Object) x;
    B.f((A1) o1); //prints true
    B.f((A2) o1); //prints false
    Object o2 = (Object) y;
    B.f((A1) o2); //prints true
    B.f(A2) o2); //prints false
    B.f((A1) null); //prints true
    B.f((A2) null); //prints false
    /* B.f(null);
      generates compile time error; ambiguous reference to f.
      null.f(new A1());
      generates compile time error; can't invoke a method on null.
    */
    BB z = null;
    /* z.f(x);
      generates a runtime error: nullpointer exception.
    */
    }
  }
)
```

7.2.2. Initialization and `instanceof`

JCFnullex1 = JCFsp ∪ `file:c.java`(

```
class P { }
class Q {
  P g;
  }
abstract class R{ }
interface I { }
class c {
  static void m() {
    co.p(null instaceof Object); //prints false
    co.p(null instaceof P); //prints false
    co.p(((P) null) instaceof Object); //prints false
    co.p(((P) null) instaceof P); //prints false
    co.p(null instaceof R); //prints false
    co.p(null instaceof I); //prints false
    P x = null;
```

```
    co.p(x instaceof P); //prints false
    P y;
    /* co.p(x == y);
      compile time error: y may not have been initialized.
    */
    Q z = new Q();
    co.p(z == null); //prints false
    co.p(z.g == null); //prints true
    co.p(z.g instance of P); //prints false
    co.p(z instance of Q); //prints true
    co.p(z instance of Object); //prints true
    /*co.p(z instanceof P);
      compile time error: impossible for z to be an instance of P.
    /*
    co.p((Object) z instanceof Q); //prints true
    co.p((Object) z instanceof P); //prints false
    co.p((Object) z instanceof R); //prints false
    co.p((Object) z instanceof I); //prints false
    y = null;
    /* co.p(y.g() == null);
      compile time error: g not found in class P.
    */
    try {
    co.p(((Q) ((Object) y)).g == null);
      /* runtime exception: the field g cannot be found:
      NullPointerException.
    */
    }catch(Exception e) {co.p(0);} //prints 0
    }
  }
)
```

Here is a program using `null` in an anonymous class. JCFnullex2 = JCFsp ∪ `file:c.java`(

```
class O {
  Object p = null;
  Object f() {return null;}
  Object g() {return null;}
  }
class c {
  final static Object p1 = new Object();
  final static Objects p2 = new Object();
  static void m() {
    O q = new O () {
      Object p = p1;
      Object f() {return p1;}
      Object g() {return p2;}
      };
    co.p(q.p == null); //prints true
    co.p(q.p == p1); //prints false
    co.p(q.f() == null); //prints false !!
    co.p(q.f() == p1); //prints true !!
    co.p(q.g() == p2); //prints true
    }
  }
)
```

7.2.3. Summary of observations

- The object `null` exists for each class. All different versions of `null` are identical in the sense of `==` provided their classes are compatible (either one extends the other).
- An object having a single field with value `null` is itself not a version of `null`.
- Method selection (overloading resolution) is sensitive for the type of a `null` argument when it occurs in the argument. (In the target position all versions of `null` generate an exception.)
- the Java operator `'- instanceof CLASS'` produces false for all versions of `null` and for each class. (Overloading resolution is more 'sensitive' than `instanceof`.)
- The default constructor will initialize fields with `null` provided no other initialization of these fields is present in the class definition.
- Selecting a field from a version of `null` will lead to a compile-time error in some cases and to a run-time error in other cases.
- If a method needs to return objects of class `X` it is correct to let it return `null`. This return can serve as a dummy content for methods delivering a class type.
- There is a significant difference between hiding (fields) and overriding (methods). An example is already found in the case of anonymous class definition using the distinction between `null` and an object of class `Object`.

Getting a complete picture of the 'behavior' of `null` is not at all an obvious matter. For most programming tasks the above summary of facts will provide sufficient information, however.

7.3. Design morphology, profile JCF's and incarnation

A program features design morphology if its code incorporates stages of its design. Design morphology is useful if intermediate stages of system development are informative and valuable, but at the same time the introduction of an independent formalism for that purpose produces an unacceptable overhead. In this text a form of design morphology will be used.

For a JCF (say JCFxyz) a profile JCF (JCFp_xyz) can be introduced. In a profile JCF all algorithmic content is removed, method bodies being empty or almost empty. Class X in JCFp_xyx is a profile for class X of JCFxyz. Conversely X in JCFxyz is called an incarnation of its profile class.

It should be noticed that the SUN JDK provides a facility called JavaDoc. It provides information very similar to our profile classes and it does so in an automatic fashion (given the incarnations as an input). The advantage of profile classes lies in the option to use the Java system for manufacturing those in advance of further design steps.

When writing profile classes the reduction of information may lead to a disconnection between different classes. Special comments are used to indicate such matters. For instance the comment `auxiliary for Y` in a profile class `X` indicates that `X` will only be used in class `Y`.

8. Molecules are aggregates having objects as parts

A molecule consists of atoms (objects) as well as some other items perhaps. Different objects in a molecule can be considered part of the molecule. The molecule itself may be considered an aggregate of objects (and may be other items). The concepts of whole (aggregate) and part are discussed in a branch of philosophy termed mereology. Nowadays mereology is a respectable part of ontology for computing, documented in a rich and growing literature.

We use Ref. [8] as a source for applied mereology (adjusted to computing purposes when needed). Two significant conclusions follow from the analysis in Ref. [8]: for a combination of objects to count as an aggregate it is mandatory that the combination enjoys at least one so-called emergent property. Emergent properties are not simply inherited from parts that have them as well. In the examples below molecules are used for representing various elementary mathematical notions. It takes a molecule to represent a number and the property of representing a natural number may be considered emergent. As a consequence, the meaningful (used) molecules each have the very property of carrying a certain meaning as an emergent property. Random molecules, however, seem to have no emergent property.

The second dogma from mereology relevant for this discussion is that the 'part of' relation is anti-symmetric. It cannot have cycles. Consider JCFagg1 below: JCFagg1 = `file:agg1.java(`

```
class A {
  B f;
  }
class B { };
class c {
  static void m() {
    A a = new A();
    B b = new B();
    a.f = b;
    }
  }
).
```

Here the object `a` defined inside method `c.m()` may be thought of as an aggregate having the object `b` as a part. As an emergent property one may take that `a` represents a

pointer. This may support the common view in Java textbooks that fields point to parts of an object. In general that view is untenable, however, because of the possible occurrence of cycles. Consider JCFagg2 = file:agg2.java(

```
Class A {
  B f;
  }
Class B {
  A g;
  }
class c{
  static void m() {
    A a = new A();
    B b = new B();
    a.f = b;
    b.g = a;
    }
  }
).
```

It follows from the concepts of mereology (in a form used in software theory) that b cannot be a part of a. Indeed on grounds of symmetry one would conclude that b is a part of a as well and a forbidden cycle emerges.

These examples are of essential importance to our proposals. It is demonstrated that molecules cannot be identified with objects (an identification which seems to be entirely common throughout the entire Java course literature). This indicates the need for a name to describe the aggregation level of molecules. This name must be different from 'object' and 'class'. Undeniably the term 'molecule' is a rather arbitrary and suggestive choice.

8.1. Molecule-oriented programming

At this point several of the previous considerations can be combined. The chemical methaphor can be turned into a programming style (molecule-oriented programming) concentrating on programs that are best understood with this methaphor in mind.

This programming style is claimed to be useful for conceptual programming. The remaining examples in this paper are meant to substantiate that claim.

As an effort in conceptual programming the three program text construction cycles mentioned before will apply. In particular the rational program text construction cycle iterates until an intelligible model of some conceptual domain has been obtained.

Although conceptual programming may not easily appeal to an engineer, its use may still be significant in view of the major difficulties encountered when formalised models of conceptual domains are designed without adequate automated support. In particular in the case that a mathematically styled model tends to be clumsy and hampered by seemingly irrelevant detail, a conceptual programming approach may still allow systematic progress leading to a model that admits fruitful communication to its readers.

If conceptual programming in Java is considered a relevant objective, the use of a molecule-oriented style is an option that cannot be neglected.

9. Natural numbers

The following class family explains the elements of arithmetic on the natural numbers in terms of Java objects. The class NNj provides a perfect class representation for all natural numbers that are needed during a computation. It is possible to represent NNm without making use of a predecessor field. In that case the predecessor of a number (represented by an object x) is found by counting from zero until an object y equal to x is found (This involves the test $x == y$.) The object from which y was found by selecting succ is then returned as the predecessor of x. We consider this representation an artificial one. It is hard to imagine that the successor is so much more primitive than the predecessor. In terms of the chemical metaphor during a computation the set of existing numbers consists of one (growing) molecule. The representation of natural numbers as in the class NNj is by no means the only reasonable option. This representation, however, has the virtue that computing the successor of a number requires a constant number of steps, which is in accordance with basic intuitions regarding the nature of numbers. If set theory is dropped as a fundamental ontology the numbers cannot be separated anymore from their representation dependent algorithmic properties.

9.1. Profile classes for NNj

The profile JCF for NNj are is JCPp_nnj = file:p_NNj.java(

```
final class NNj {
  static NNj z;
    //focus to 0
  synchronized NNj S() {return null;};
    //returns a focus to the successor of
    the target
  NNj P() {returns null;};
    //returns a focus to the predecessor
    of the target
  }
class enrichNNj {
  static NNj zero;
    //focus to 0
  static NNj one;
```

```
    //focus to 1
  NNj S(NNj x) {return null;};
    //Successor function in prefix notation
  static NNj P(NNj x) {return null;};
    //Predecessor function in prefix notation
  static NNj plus (NNj x, NNj y) {return null:};
    //addition
  stastic NNj monus (NNj x, NNj y) {return null:};
    //cut-off substration
  static NNj mult(NNj x, NNj y) {return null:};
    //multiplication
  }
)
```

In JCFp_NNj the successor and predecessor operators act in postfix style on their argument (target). Further the almost empty body for non-void methods returns `null`. This is a necessity due to the Java syntax, as something has to be returned.[2] In the enriching class, versions of these operators are provided acting in the usual infix (prefix) style. The subtraction operator is termed `monus` (rather than `minus`) following the convention in logic for a subtraction returning 0 where a negative result is expected.

9.2. Incarnation classes for NNj

Incarnations of these profile classes are given in JCFnnj = JCFsp ∪ JCFp4nnj ∪ `file:NNj.java`(

```
final class NNj {
  //Each number will be given by an object in a molecule. The
  //molecule has the form of a doubly linked list, s arrows going
  //upwards and p arrows going downwards. Focus z will contain
  //the first object (representing 0), and at any time there is a
  //last object for which the field s contains null. The p field of
  //0 contains null all the time.
  static final NNj z = new NNj ();
  private NNj s;
    //contains the successor (except at 0)
  private NNj s;
    //contains the successor (except at the current maximum)
  private NNj ();
    //makes the constructor inaccessible from outside
  private NNj newS(); {NNj x = new NNj (); x.p = this; s = x; return x;}
    //molecule is extended with one atom which serves as the
    //successor (of its last object). The new atom is returned after
    //the p and s fields have been adjusted. Auxiliary for S().
  synchronized NNj S() {return (s == null)? newS():s;}
  NNj P() (return (p == null)? z:p;}
  }
class enrichNNj {
//An abstract data type using functional (prefix) notation.
//Function definitions based on well-known recursion equations.
static NNj zero = NNj.z;
static NNj S(NNj x) {return x.S();}
static NNj one = S(zero);
```

[2] If profiles (profile classes) were native to Java a relaxation of this requirement (like in the case of abstract methods) would be very helpful.

```
  static NNj P(NNj x) {return x.P();}
  static NNj plus(NNj x, NNj y) {return
    (y == zero)? x:S(plus(x,P(y)));}
  static NNj monus(NNj x, NNj y) {return
    (y == zero)? x:P(monus(x,P(y)));}
  static NNj multi(NNj x, NNj y) {return
    (y == zero)? zero:plus(multi(x,P(y)));}
  }
  )
```

9.3. A test class for NNj

A test is performed in JCFnnjtest = JCFnnj ∪ `file:c.java`(

```
  class c extends enrichNNj {
    /*Some test cases*/
    static void m() {
      boolean b = (S(P(P(S(zero)))) == P(P(S(one))));
      co. p(b);
      NNj x = plus(S(one), S(one));
      NNj y == plus(S(x), S(x));
      b = (P(P(P(P(y))))) = S(S(S(S(S(zero))))));
      co.p(b);
      NNj u = mult(one, S(one));
      NNj v = plus(one, one);
      B = (u == v);
      Co.p(b);
      U = multi(x,S(one));
      V = Plus(x,x);
      b = (u == v);
      co.p(b);
      }
    }
  /*Generated output when running s:
    false
    true
    true
    true
  */
  )
```

10. MFCP

In order to facilitate programming at a larger scale it is almost indispensible to collect class text files and class files in packages. Otherwise the sharing of potentially reusable classes with other programmers becomes problematic, unless one accepts the version control problems emerging from distribution by means of file copying.

We will collect various classes related to molecule-oriented programming in a package named MFCP (Molecule-oriented Foundation Classes Package).

Packages are quite demanding on their contents. Source texts containing classes used from outside the package need to be placed in their own file carrying the same name. Each source file in the package (MFCP) needs to start with the corresponding package statement (`package MFCP;`). Further each element that needs to be used in another package must be made public. If the package is used it must be ensured that those class files belonging to the package that are actually needed in the application are contained in a subdirectory (named with the package name) of a directory listed in the class path. Multiple occurrences of

these files in different directories along the CLASSPATH may lead to complications, however. Indeed, the package may be split up over different directories, but the contents of the different parts must be file collections with disjoint name sets.

The introduction of NNj in MFCP can be achieved as follows:

JCFnnip = folder:MFCP(file:NNj.java(

```
package MFCP;
public final class NNj {
  public static final NNj z = new NNj();
  private NNj p,s;
  private NNj () { }
  private NNj news() {NNj x = new NNj();
    x.p = this; s = x; return x;}
  public synchronized NNj S() {return (s == null)? news():s;}
  public NNj P() {return (p == null)? z:p;}
  }
) ∪ file: enrichNNj.Java(
package MFCP;
public class enrichNNj {
  public static NNj zero = NNj.z;
  public static NNj S(NNj x) {return x.S();}
  public static NNj one = S(zero);
  public static NNj P(NNj x) {return x.P();}
  public static NNj plus(NNj x, NNj y){return
    (y == zero)? x:S(plus(x, P(y)));}
  public static NNj monus(NNj x, NNj y){return
    (y == zero)? x:P(plus(x, P(y)));}
  public static NNj multi(NNj x, NNj y){return
    (y == zero)? zero:plus(multi(x, P(y)));}
  }
)).
```

In order to make the test program work it needs to be preceded with `import MFCP.` or with `import MFCPjab.NNj;import MFCP.enrichNNj;`. In both cases the package contents should be available via the CLASSPATH.

10.1. Integral numbers

Java objects and molecules for integers (INj for Integral Numbers in Java) are introduced in the following class family. Profile classes are omitted (but may easily be defined by the reader). Incarnations are in: JCFinjp = JCFsp ∪ folder:MFCP(file:SIGNj.java(

```
package MFCP;
public class SIGNj {
  public static final SIGNJ neg = new SIGNj();
  public static final SIGNJ pos = new SIGNj();
  }
) ∪ file:INj.Java(
package MFCP;
public final class INj
  //A perfect class for the integers.
  extends SIGNj {
  public static final INj z = new INj();
  private INj s,p;
  private SIGNj sign;
  private INj() { }
```

```
    private INj s() {INj x = new INj();
      x.p = this; s = x; x.sign = pos;
      return x;}
    private INj s() {INj x = new INj();
      x.s = this; p = x; x.sign = neg;
      return x;}
    public INj S() {return(s == null)?s():s;}
    public INj S() {return(p == null)?p():p;}
    public SIGNj sign() {return sign;}
    }
) ∪ file.viewINj.java(
package MFCP;
public class viewINj extends SIGNj {
    //transition to prefix notation.
    public static final INj zero = INj.z;
    public static SIGNj sign(INj x) {return x.sign();}
    public static INj S(INj x) {return x.S();}
    public static INj P(INj x) {return x.P();}
    }
) ∪ file.enrichINj.java(
package MFCP;
public class enrichINj extends viewINj {
    //contains major arithmetic operator definitions
    public static INj one = S(zero);
    public static INj min (INj x){
      if(x == zero) {return x;}
      else {return(sign(x) == pos)?
        P(min(P(x))): S(min(S(x)));}}
    public static INj plus(INj x, INj y) {
      if (y == zero) {return x;}
      else {return (sign(y) == pos)?
        plus(S(x),P(y)):plus(P(x),S(y));}
      }
    public static INj minus(INj x, INj y) {
      return plus (x, min(y));}
    public static INj mult(INj x, Inj y){
      if (y == zero) {return zero;}
      else {return (sign(y) == pos)?
        plus(mult(x,P(y)),x):
        minus(mult(x,S(y)),x);}
      }
    }
))
```

10.2. Cartesian classes

There is a need to have explicit terminology regarding the connection between Java classes on the one hand and the sets of Math objects represented by Java classes on the other hand. The notion of a perfect class representation (in brief a perfect class, assuming it is obvious what is to be represented) is a class where object identity (==) corresponds to the most reasonable form of Math object equality.

Obviously in a perfect class it is implausible to introduce an `equals(-)` method because there is no more plausible equality than == which is a built-in Java feature already.

When dealing with Cartesian products the situation is different. For a Math object $\langle a, b\rangle$ representing a pair the most plausible equality is the logical conjunction of the equality of components. In general (see the class `NNxNNj`) this is not object

identity because in order to achieve that a meticulous bookkeeping of all objects that have been constructed during a computation is needed. Such a bookkeeping may well be far too expensive in terms of computation time.[3]

A class is called Cartesian if the most plausible equality on its objects is pairwise equality of components. Here the components of object a are all objects (and values) found after selection of the various instance fields of a. More specifically: a class is called Cartesian if it has a boolean method equals(-) returning true on a call a.equals(b) if either a == b or a and b have equal components. This raises the question which form of equality is used on the various component classes. If a component class contains an equals(-) method that method is used; otherwise object identity (==) is used.

If a class is not Cartesian it is called non-Cartesian. The class SMj below is a non-Cartesian class. The best form of equality on it identifies two points if there exists a graph isomorphism in the object graph permuting the two objects. That being out of the question in terms of Java programming no attempt to introduce a more interesting form of equality is made.

10.3. Complex integers

The Gaussian ring contains the integral points in the complex plane. This structure is specified in the next class. JCFcinjp = JCFsp ∪ folder:MFCP(file:CINj.java(

```
package MFCP;
public class CINj
  extends enrichINj {//A Cartesian class for the Gaussian ring.
  private INj re;
  private INj im;
  public CINj(INj x, INj y) {re = x, im = y;}
  public Boolean equals (CINj x, CINj y) {return
    ((re(x) == re(y)&&(im(x) == im(y)));
    }
  public static INj re(CINj x) (return x.re}
  public static INj im(CINj x) (return x.im}
  public static CINj plus(CINj x, CINj y){return
    new CINj(plus(re(x), re(y)), plus(im(x), im(y)));
    }
  public static CINj minus(CINj x, CINj y){return
    new CINj(minus(re(x), re(y)), minus(im(x), im(y)));
    }
  public static CINj mult(CINj x,CINj y) {return
    new CINj(
      minus(mult(re(x),re(y)), mult(im(x), im(y)),
      plus(mult(re(x), im(y)), mult(im(x), re(y)))
    );
    }
  }
))
```

10.4. Collecting the package MFCP

Thus far the following part of MFCP has been developed: JCFmfcnic = JCFnnj ∪ JCFinjp ∪ JCFcinjp.

At this stage many questions may be posed regarding the adequacy of this small package. We mention some points:

The relation between NNj and INj is quite weak. Can this be improved?

Conversion methods between NNj to INj may be introduced, or a substantial use of inheritance can be made? Should it be the case that INj is a subclass of NNj, and CINj is a subclass of INj, or just the other way around?

(Our current opinion, based on a far larger body of examples than presented above, indicates that Java imposes definite

[3] If the objects are very big, however, there may be a solid advantage in preventing object duplication. This is exacltly the virtue of the Aterm library for Asf + Sdf.

limitations on the modular structure of classes appearing in a molecule-oriented programming project. The inclusions of extensions and/or formal class extensions cannot easily be forced to coincide with the classification of objects one may have in mind. The very absence of multiple inheritances poses a major problem, as multiple inheritances seem a perfectly natural way of thinking about classification of objects.)

Can these programs be simplified (significantly)?

(In the case of NNj so much time has been spent to arrive at a simplest program, that the author would be suprised if a significant simplification emerges. Proving the non-existence of such a simplification is not as feasible task at the moment, however.)

Will the naming convention used here be workable on the long run? (Names have been kept quite short on purpose, thus deviating from more common naming schemes in use for Java.)

Is the use of so many static methods justified?

(This use emerged after long experimentation with instance methods. The asymmetry of such programs was a cause for rejection, however. For that reason the use of static methods for the purpose of conceptual programming is considered justified.)

11. Conclusions

Conceptual programming has been proposed as a realistic ambition for programmers. The parts of the software life-cycle relevant for conceptual programming have been identified. Molecule-oriented programming (in Java) has been proposed as a programming style fruitful for conceptual programming. In addition it has been argued that molecule-oriented programming provides a clear mental model of the execution (memory state evolution) of an object-oriented program in Java.

A series of conceptual Java programs, written in molecule-oriented style, has been presented. These programs provide stand alone definitions (Java presented theories) of several elementary but important number algebras.

In addition several auxiliary methods and techniques have been investigated: empirical semantics (for Java), the Java class family notation (allowing a systematic presentation of multiclass Java programs in a purely descriptive setting), and profile classes (serving as a remedy against the limitations of Java's interface concept).

Appendix A

Folder hierarchy notation

FHN allows the notation of named and structured collections of files. A FHN expression may contain (describe) a JCF. A sequence of FHN notations may encode a portfolio of JCF's. FHN provides a syntax for the description of folders. Folders can be empty, or they can contain a number of named files/folders, files being organized sequentially, folders hierarchically. In the detailed description below A, A_i range over texts and B, B_i range over folders. The size of a folder is its number of elements at the top level. In the case of JCF texts A, A_i are chosen to be flat file representations of classes. Concatenation of texts is denoted with '$*$'.

The primitives of FHN are these:

- $\emptyset$ represents the empty folder. Its size is 0.
- `file:` *fileName*(A), denotes a file named *fileName* containing the character sequence A. The file `file:` *fileName*(A) is itself a folder of size 1.
- `folder:` *folderName* (B), denotes a folder named *folderName* containing the folder B. The folder `folder:` *folderName*(B) has size 1.
- Given folders collection B_1 and B_2, $B_1 \cup B_2$ is the union of the two folders. There are some constraints:

  In this union the folders of B_1 that have names also occurring as B_2 names are united. For a particular name a folder contains at most one folder with that name.

  The situation for files is similar but not identical. If B_1 contains a file `file:`*fileName* (A_1) and B_2 contains a file `file:`*fileName*(A_2), the combination $B_1 \cup B_2$ will contain a file `file:`*fileName*$(A_1 * A_2)$, ($*$ denoting string concatenation.) This rule prevents folder union from being commutative. Again a folder can contain (at each level) at most one file under some given name.

An FHN-Java expression is an FHN-expression with all texts A listed in files being class descriptions. For such an expression to be syntactically correct the empirical criterion is that it is accepted by the compiler of JDK 1.2.1 (or later). FHN-expressions for Java will be used to represent JCF's.

A path is a sequence of names written as $name_1/name_2/\ldots/name_k$. Paths can be used to indicate files of folders within an FHN-expression or a JCF. With B a JCF and q a path $B.\texttt{file}(p)$ denotes the text (if any) within B with path q leading to it. Similarly $B.\texttt{folder}(p)$ denotes the folder (if any) reached along path q. Thus `file:`*fileName*(A).`file`(*fileName*) $= A$ etc. If no file name or folder can be found using the selection operator the field.folder is produced.

In addition to the constructors and selectors mentioned above FHN admits a subtraction operator $\Gamma - \{p_1, \ldots, p_k\}$ removing all files and folders named in the set $\{p_1, \ldots, p_k\}$ from the JCF Γ.

A.1. Equations for FHN

The rules for the folder hierarchy notation FHN can easily be summarized in a set of conditional equations.[4] The use of the equations is as follows: in the process of writing a JCF, named files and named folders can be introduced in successive stages. The equations describe how to combine parts of files and folders into complete files and folders. This process has to be performed before deletion of named folders or files is attempted. The flexibility of FHN is helpful when a portfolio of JCF's is to be denoted. In these equations x,y range over names of files/folders, A, A_i range over texts (file contents), B, B_i range over folders, p, p_i range over paths.

A.2. Equations for folders

$$\emptyset \cup B = B, B \cup \emptyset = B$$

$$(B_1 \cup B_2) \cup B_3 = B_1 \cup (B_2 \cup B_3)$$

$$\texttt{file} : x(A_1) \cup \texttt{file} : x(A_2) = \texttt{file} : x(A_1 * A_2)$$

$$\texttt{folder} : x(B_1) \cup \texttt{folder} : x(B_2) = \texttt{folder} : x(B_1 \cup B_2)$$

$$\texttt{folder} : x(B) \cup \texttt{file} : x(A) = \texttt{folder} : x(B)$$

$$x \neq y \rightarrow \texttt{folder} : x(B_1) \cup \texttt{folder} : y(B_2) = \texttt{folder} : y(B_2) \cup \texttt{folder} : x(B_1)$$

$$x \neq y \rightarrow \texttt{file} : x(A) \cup \texttt{folder} : y(B) = \texttt{folder} : y(B) \cup \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{file} : x(A_1) \cup \texttt{file} : y(A_2) = \texttt{file} : y(A_2) \cup \texttt{file} : x(A_1)$$

A.3. Equations for the remove operator

The equations for the remove operator can be defined on top of this basis (q ranges over paths):

$$B - \{p_1, \ldots, p_{n+1}\} = (B - \{p_1\}) - \{p_2, \ldots, p_{n+1}\}$$

$$(B_1 \cup B_2) - \{p\} = (B_1 - \{p\}) \cup (B_2 - \{p\})$$

$$\emptyset - \{p\} = \emptyset$$

$$x \neq y \rightarrow \texttt{file} : x(A) - \{y\} = \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{folder} : x(B) - \{y\} = \texttt{folder} : x(B)$$

$$\texttt{file} : x(A) - \{x\} = \emptyset$$

$$\texttt{folder} : x(B) - \{x\} = \emptyset$$

$$\texttt{file} : x(A) - \{y/p\} = \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{folder} : x(B) - \{y/p\} = \texttt{folder} : x(B)$$

$$\texttt{folder} : x(B) - \{x/p\} = \texttt{folder} : x(B - \{p\})$$

A.4. Defining equations for the selectors

The equations for both selection operators can be given, using the remove operator. Four equations describe the 'positive cases'.

A.4.1. Positive cases

$$(\texttt{file} : x(A) \cup (B - \{x\})).\texttt{file}(x) = A$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{folder}(x) = B_1$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{file}(x/p) = B_1.\texttt{file}(p)$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{folder}(x/p) = B_1.\texttt{folder}(p)$$

A.4.2. Negative cases

In all other cases an error is to be produced (represented by M.). This requires a disappointingly large number of defining equations:

$$\emptyset.\texttt{file}(x) = M$$

$$\emptyset.\texttt{folder}(x) = M$$

$$(B - \{x\}).\texttt{file}(x) = M$$

$$(B - \{x\}).\texttt{file}(x/p) = M$$

$$(B - \{x\}).\texttt{folder}(x) = M$$

$$(B - \{x\}).\texttt{folder}(x/p) = M$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{file}(x) = M$$

$$(\texttt{file} : x(A) \cup (B - \{x\})).\texttt{folder}(x) = M$$

[4] FHN is called a notation in spite of its being an algebra according to even the most restricted definitions. Although FHN is an algebra, its equations are neither deep nor difficult. The transformation rules expressed by the equations are all fairly obvious. FHN is a bookkeeping device and no more than that. This lack of intrinsic interest of the FHN equations motivates the decision to call it a notation rather than an algebra or a calculus.

References

[1] D.A. Bailey, Data Structures in Java for the Principled Programmer, McGraw-Hill, Singapore, 1999.

[2] J.A. Bergstra, J.-W. Klop, Process algebra for synchronous communication, Information and Control 60 (1/3) (1984) 109–137.

[3] J.A. Bergstra, M.E. Loots, Empirical semantics for object-oriented

programs. Technical Report 007, Department of Philosophy, Utrecht University, (1999).
[4] J.A. Bergstra, M.E. Loots, Program algebra for component code, Formal Aspects of Computing 12 (2000) 1–17.
[5] J.W. de Bakker, Mathematical Theory of Program Correctness, Prentice-Hall International, Mountain View, CA, 1980.
[6] M. Felleisen, D.P. Friedman, A Little Java a Few Patterns, Sun Soft Press, Prentice Hall International, Mountain View, CA, 1996.
[7] J. Heering, P. Klint, Semantics of programming languages: a tool oriented approach, Technical Report R9920, CWI Amsterdam, SEN, 1999. ACM COR preprint server cs. PL/9911001 v2.
[8] A.L. Opdahl, B. Henderson-Sellers, F. Barbier, Ontological analysis of whole-part relationships in OO-models, Information and Software Technology 43 (2001) 387–399.
[9] I. Reymen, Improving Design Processes Through Structured Reflection, IPA Dissertation Series 2001–4, University Press, Technical University Eindhoven, Eindhoven, 2001.
[10] J. Skansholm, Java From the Beginning, Addisson-Wesley, Reading, MA, 1999.