Programming Paradigms for Dummies: What Every Programmer Should Know

Peter Van Roy

This chapter gives an introduction to all the main programming paradigms, their underlying concepts, and the relationships between them. We give a broad view to help programmers choose the right concepts they need to solve the problems at hand. We give a taxonomy of almost 30 useful programming paradigms and how they are related. Most of them differ only in one or a few concepts, but this can make a world of difference in programming. We explain briefly how programming paradigms influence language design, and we show two sweet spots: dual-paradigm languages and a definitive language. We introduce the main concepts of programming languages: records, closures, independence (concurrency), and named state. We explain the main principles of data abstraction and how it lets us organize large programs. Finally, we conclude by focusing on concurrency, which is widely considered the hardest concept to program with. We present four little-known but important paradigms that greatly simplify concurrent programming with respect to mainstream languages: declarative concurrency (both eager and lazy), functional reactive programming, discrete synchronous programming, and constraint programming. These paradigms have no race conditions and can be used in cases where no other paradigm works. We explain why for multi-core processors and we give several examples from computer music, which often uses these paradigms.

More is not better (or worse) than less, just different.
– The paradigm paradox.

1 Introduction

Programming is a rich discipline and practical programming languages are usually quite complicated. Fortunately, the important ideas of programming languages are simple. This chapter is intended to give readers with some programming experience a running start for these ideas. Although we do not give formal definitions, we give precise intuitions and good references so that interested readers can quickly get started using the concepts and languages that implement them. We mention *all* important paradigms but we favor some little-known paradigms that deserve to be more widely used. We have deliberately left out detailed explanations of some of the more well-known paradigms

(such as functional and object-oriented programming), since they already have a huge literature.

Solving a programming problem requires choosing the right concepts. All but the smallest toy problems require different sets of concepts for different parts. This is why programming languages should support many paradigms. A *programming paradigm* is an approach to programming a computer based on a mathematical theory or a coherent set of principles. Each paradigm supports a set of concepts that makes it the best for a certain kind of problem. For example, object-oriented programming is best for problems with a large number of related data abstractions organized in a hierarchy. Logic programming is best for transforming or navigating complex symbolic structures according to logical rules. Discrete synchronous programming is best for reactive problems, i.e., problems that consist of reactions to sequences of external events. Languages that support these three paradigms are respectively Java, Prolog, and Esterel.

Popular mainstream languages such as Java or C++ support just one or two separate paradigms. This is unfortunate, since different programming problems need different programming concepts to solve them cleanly, and those one or two paradigms often do not contain the right concepts. A language should ideally support many concepts in a well-factored way, so that the programmer can choose the right concepts whenever they are needed without being encumbered by the others. This style of programming is sometimes called *multiparadigm* programming, implying that it is something exotic and out of the ordinary. On the contrary, in our experience it is clear that it should be the normal way of programming. Mainstream languages are far from supporting this. Nevertheless, understanding the right concepts can help improve programming style even in languages that do not directly support them, just as object-oriented programming is possible in C with the right programmer attitude.

This chapter is partly based on the book [50], familiarly known as CTM, which gives much more information on many of the paradigms and concepts presented here. But this chapter goes further and presents ideas and paradigms not covered in CTM. The code examples in this chapter are written in the Oz language, which is also used in CTM. Oz has the advantage that it supports multiple paradigms well, so that we do not have to introduce more than one notation. The examples should be fairly self-explanatory; whenever anything out of the ordinary occurs we explain it in the text.

Contents of this chapter

Languages, paradigms, and concepts Section 2 explains what programming paradigms are and gives a taxonomy of the main paradigms. If your experience is limited to one or just a few programming languages or paradigms (e.g., object-oriented programming in Java), then you will find a much broader viewpoint here. We also explain how we organize the paradigms to show how they are related. We find that it is certainly not true that there is one "best" paradigm, and a fortiori this is not object-oriented programming! On the contrary, there are many useful paradigms. Each paradigm has its place: each has problems for which it gives the best solution (simplest, easiest to reason about, or most efficient). Since most programs have to solve more than one problem, it follows that they are best written in different paradigms.

Designing a language and its programs Section 3 explains how to design languages to support several paradigms. A good language for large programs must support several paradigms. One approach that works surprisingly well is the *dual-paradigm* language: a language that supports one paradigm for programming in the small and another for programming in the large. Another approach is the idea of designing a *definitive* language. We present an example design that has proved itself in four different areas. The design has a layered structure with one paradigm in each layer. Each paradigm is carefully chosen to solve the successive problems that appear. We explain why this design is good for building large-scale software.

Programming concepts Section 4 explains the four most important concepts in programming: records, lexically scoped closures, independence (concurrency), and named state. Each of these concepts gives the programmer an essential expressiveness that cannot be obtained in any other way. These concepts are often used in programming paradigms.

Data abstraction Section 5 explains how to define new forms of data with their operations in a program. We show the four kinds of data abstractions: objects and abstract data types are the two most popular, but there exist two others, declarative objects and stateful abstract data types. Data abstraction allows to organize programs into understandable pieces, which is important for clarity, maintenance, and scalability. It allows to increase a language's expressiveness by defining new languages on top of the existing language. This makes data abstraction an important part of most paradigms.

Deterministic concurrent programming Section 6 presents deterministic concurrent programming, a concurrent model that trades expressiveness for ease of programming. It is *much* easier to program in than the usual concurrent paradigms, namely shared-state concurrency and message-passing concurrency. It is also by far the easiest way to write parallel programs, i.e., programs that run on multiple processors such as multi-core processors. We present three important paradigms of deterministic concurrency that deserve to be better known. The price for using deterministic concurrency is that programs cannot express nondeterminism, i.e., where the execution is not completely determined by the specification. For example, a client/server application with two clients is nondeterministic since the server does not know from which client the next command will come. The inability to express nondeterminism inside a program is often irrelevant, since nondeterminism is either not needed, comes from outside the program, or can be limited to a small part of the program. In the client/server application, only the communication with the server is nondeterministic. The client and server implementations can themselves be completely deterministic.

Constraint programming Section 7 presents the most declarative paradigm of our taxonomy, in the original sense of declarative: telling the computer what is needed instead of how to calculate it. This paradigm provides a high level of abstraction for solving problems with global conditions. This has been used in the past for combinatorial problems, but it can also be used for many more general applications such as computer-aided composition. Constraint programming has achieved a high degree of maturity since its

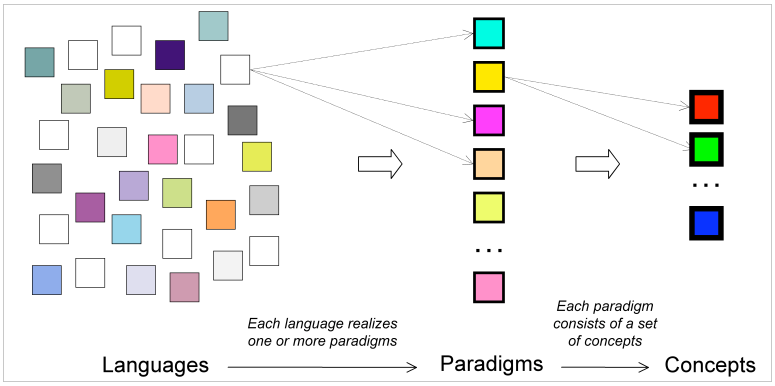

Figure 1. Languages, paradigms, and concepts

origins in the 1970s. It uses sophisticated algorithms to find solutions that satisfy global conditions. This means that it genuinely delivers on its ambitious claims.

Conclusions and suggestions for going further Section 8 concludes by reiterating why programming languages should support several paradigms. To understand the "soul" of each paradigm and to gain experience programming with different paradigms, we recommend the use of a multiparadigm language. A multiparadigm language permits programming in each paradigm without interference from other paradigms. The two most extensive multiparadigm languages are the dynamically typed language Oz [50] and the statically typed language Alice [38].

2 Languages, paradigms, and concepts

This section gives the big picture of programming paradigms, the languages that realize them, and the concepts they contain. There are many fewer programming paradigms than programming languages. That is why it is interesting to focus on paradigms rather than languages. From this viewpoint, such languages as Java, Javascript, C#, Ruby, and Python are all virtually identical: they all implement the object-oriented paradigm with only minor differences, at least from the vantage point of paradigms.

Figure 1 shows the path from languages to paradigms and concepts. Each programming language realizes one or more paradigms. Each paradigm is defined by a set of programming concepts, organized into a simple core language called the paradigm's *kernel language*. There are a huge number of programming languages, but many fewer paradigms. But there are still a lot of paradigms. This chapter mentions 27 different paradigms that are actually used. All have good implementations and practical applications. Fortunately, paradigms are not islands: they have a lot in common. We present a taxonomy that shows how paradigms are related.

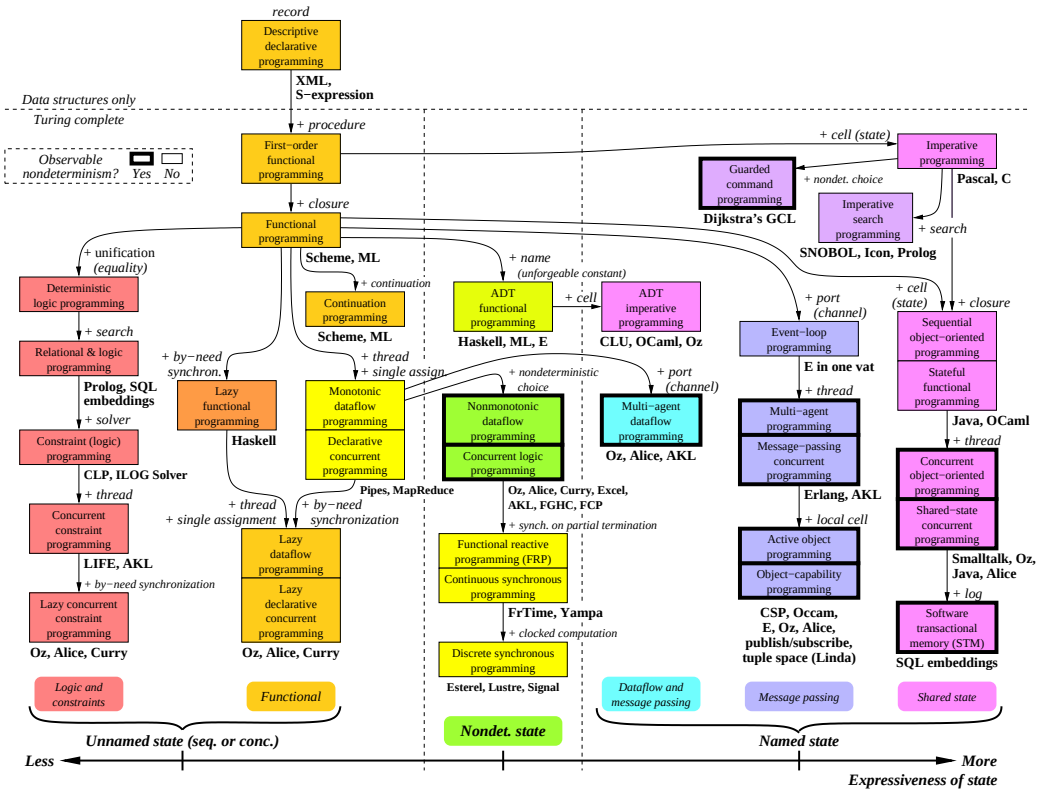

Figure 2. Taxonomy of programming paradigms

2.1 Taxonomy of programming paradigms

Figure 2 gives a taxonomy of all major programming paradigms, organized in a graph that shows how they are related [55]. This figure contains a lot of information and rewards careful examination. There are 27 boxes, each representing a paradigm as a set of programming concepts. Of these 27 boxes, eight contain two paradigms with different names but the same set of concepts. An arrow between two boxes represents the concept or concepts that have to be added to go from one paradigm to the next. The concepts are the basic primitive elements used to construct the paradigms. Often two paradigms that seem quite different (for example, functional programming and object-oriented programming) differ by just one concept. In this chapter we focus on the programming concepts and how the paradigms emerge from them. With n concepts, it is theoretically possible to construct 2^n paradigms. Of course, many of these paradigms are useless in practice, such as the empty paradigm (no concepts)[1] or paradigms with only one concept. A paradigm almost always has to be Turing complete to be practical. This explains why functional programming is so important: it is based on the concept of first-class function,

[1]Similar reasoning explains why Baskin-Robbins has exactly 31 flavors of ice cream. We postulate that they have only 5 flavors, which gives $2^5 - 1 = 31$ combinations with at least one flavor. The 32^{nd} combination is the empty flavor. The taste of the empty flavor is an open research question.

or *closure*, which makes it equivalent to the λ-calculus which is Turing complete. Of the 2^n possible paradigms, the number of practically useful paradigms is much smaller. But it is still much larger than n.

When a language is mentioned under a paradigm in Figure 2, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment. There are just too many good languages to mention them all.

Figure 2 shows two important properties of the paradigms: whether or not they have observable nondeterminism and how strongly they support state. We now discuss each of these properties in turn.

Observable nondeterminism

The first key property of a paradigm is whether or not it can express observable nondeterminism. This is identified in Figure 2 by boxes with a heavy or light border. We recall that nondeterminism is when the execution of a program is not completely determined by its specification, i.e., at some point during the execution the specification allows the program to choose what to do next. During the execution, this choice is made by a part of the run-time system called the *scheduler*. The nondeterminism is *observable* if a user can see different results from executions that start at the same internal configuration. This is highly undesirable. A typical effect is a *race condition*, where the result of a program depends on precise differences in timing between different parts of a program (a "race"). This can happen when the timing affects the choice made by the scheduler. But paradigms that have the power to express observable nondeterminism can be used to model real-world situations and to program independent activities.

We conclude that observable nondeterminism should be supported only if its expressive power is needed. This is especially true for concurrent programming. For example, the Java language can express observable nondeterminism since it has both named state and concurrency (see below). This makes concurrent programming in Java quite difficult [29]. Concurrent programming is much easier with the declarative concurrent paradigm, in which all programs are deterministic. Sections 6 and 7 present four important concurrent paradigms that do not have observable nondeterminism.

Named state

The second key property of a paradigm is how strongly it supports state. State is the ability to remember information, or more precisely, to store a sequence of values in time. Its expressive power is strongly influenced by the paradigm that contains it. We distinguish three axes of expressiveness, depending on whether the state is unnamed or named, deterministic or nondeterministic, and sequential or concurrent. This gives eight combinations in all. Later in this chapter we give examples of many of these combinations. Not all of the combinations are useful. Figure 3 shows some useful ones arranged in a lattice;

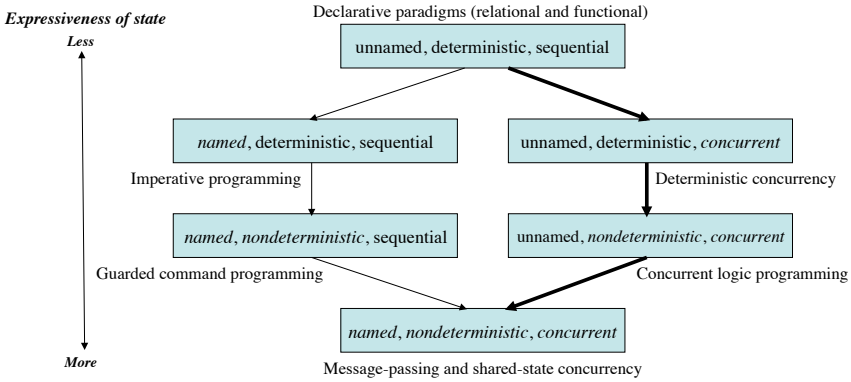

Figure 3. Different levels of support for state

adjacent boxes differ in one coordinate.[2] One intriguing box shown is Dijkstra's guarded command language (GCL) [14]. It has named state and nondeterministic choice in a sequential language. It uses nondeterministic choice to avoid overspecifying algorithms (saying too much about how they should execute).

The paradigms in Figure 2 are classified on a horizontal axis according to how strongly they support state. This horizontal axis corresponds to the bold line in Figure 3. Let us follow the line from top to bottom. The least expressive combination is functional programming (threaded state, e.g., DCGs in Prolog and monads in functional programming: unnamed, deterministic, and sequential). Adding concurrency gives declarative concurrent programming (e.g., synchrocells: unnamed, deterministic, and concurrent). Adding nondeterministic choice gives concurrent logic programming (which uses stream mergers: unnamed, nondeterministic, and concurrent). Adding ports or cells, respectively, gives message passing or shared state (both are named, nondeterministic, and concurrent). Nondeterminism is important for real-world interaction (e.g., client/server). Named state is important for modularity (see Section 4.4).

Both observable nondeterminism and named state are cases where it is important to choose a paradigm that is expressive enough, but not too expressive (see epigram at the head of the chapter). Each of these two concepts is sometimes needed but should be left out if not needed. The point is to pick a paradigm with just the right concepts. Too few and programs become complicated. Too many and reasoning becomes complicated. We will give many examples of this principle throughout this chapter.

2.2 Computer programming and system design

Figure 4 gives a view of computer programming in the context of general system design. This figure adds computer programming to a diagram taken from Weinberg [56]. The two axes represent the main properties of systems: complexity (the number of basic interacting components) and randomness (how nondeterministic the system's behavior is). There are two kinds of systems that are understood by science: aggregates (e.g., gas

[2]Two of the eight possible combinations are not shown in the figure. We leave it to the reader to discover them and find out if they make any sense!

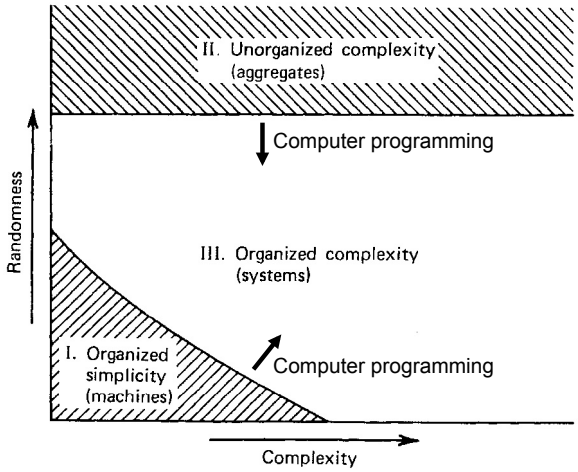

Figure 4. Computer programming and system design (adapted from Weinberg [56])

molecules in a box, understood by statistical mechanics) and machines (e.g., clocks and washing machines, a small number of components interacting in mostly deterministic fashion). The large white area in the middle is mostly not understood. The science of computer programming is pushing inwards the two frontiers of system science: computer programs can act as highly complex machines and also as aggregates through simulation. Computer programming permits the construction of the most complex systems.

Modern programming languages have evolved over more than five decades of experience in constructing programmed solutions to complex, real-world problems. Modern programs can be quite complex, reaching sizes measured in millions of lines of source code, written by large teams of programs over many years. In our view, languages that scale to this level of complexity are successful in part because they model some essential factors of how to construct complex systems. In this sense, these languages are not just arbitrary constructions of the human mind. They explore the limits of complexity in a more objective way. We would therefore like to understand them in a scientific way, i.e., by understanding the basic concepts that compose the underlying paradigms and how these concepts are designed and combined. This is the deep justification of the creative extension principle explained below.

2.3 Creative extension principle

Concepts are not combined arbitrarily to form paradigms. They can be organized according to the *creative extension principle*. This principle was first defined by Felleisen [18] and independently rediscovered in [50]. It gives us a guide for finding order in the vast set of possible paradigms. In a given paradigm, it can happen that programs become complicated for technical reasons that have no direct relationship to the specific problem that is being solved. This is a sign that there is a new concept waiting to be discovered. To show how the principle works, assume we have a simple sequential functional programming paradigm. Then here are three scenarios of how new concepts can be discovered and added to form new paradigms:

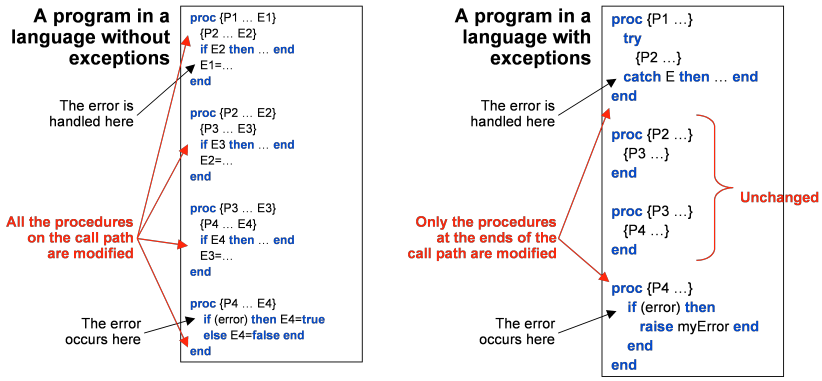

Figure 5. How adding exceptions to a language can simplify programs

- If we need to model several independent activities, then we will have to implement several execution stacks, a scheduler, and a mechanism for preempting execution from one activity to another. All this complexity is unnecessary if we add one concept to the language: *concurrency*.

- If we need to model updatable memory, that is, entities that remember and update their past, then we will have to add two arguments to all function calls relative to that entity. The arguments represent the input and output values of the memory. This is unwieldy and it is also not modular because the memory travels throughout the whole program. All this clumsiness is unnecessary if we add one concept to the language: *named state*.

- If we need to model error detection and correction, in which any function can detect an error at any time and transfer control to an error correction routine, then we need to add error codes to all function outputs and conditionals to test all function calls for returned error codes. All this complexity is unnecessary if we add one concept to the language: *exceptions*. Figure 5 shows how this works.

The common theme in these three scenarios (and many others!) is that we need to do pervasive (nonlocal) modifications of the program in order to handle a new concept. If the need for pervasive modifications manifests itself, we can take this as a sign that there is a new concept waiting to be discovered. By adding this concept to the language we no longer need these pervasive modifications and we recover the simplicity of the program. The only complexity in the program is that needed to solve the problem. No additional complexity is needed to overcome technical inadequacies of the language. Both Figure 2 and [50] are organized according to the creative extension principle.

3 Designing a language and its programs

A programming language is not designed in a vacuum, but for solving certain kinds of problems. Each problem has a paradigm that is best for it. No one paradigm is best for all problems. That is why it is important to choose carefully the paradigms supported by the

language. We will look at two interesting cases: languages that support two paradigms (Section 3.1) and layered languages (Section 3.2). The layered language we present is a particularly interesting one because almost the same layered structure appears in four different areas.

3.1 Languages that support two paradigms

Many languages support two paradigms, typically one for programming in the small and another for programming in the large. The first paradigm is chosen for the kind of problem most frequently targeted by the language. The second paradigm is chosen to support abstraction and modularity and is used when writing large programs. Here are a few examples:

- *Prolog*: The first paradigm is a logic programming engine based on unification and depth-first search. The second paradigm is imperative: the assert and retract operations which allow a program to add and remove program clauses. Prolog dates from 1972, which makes it an old language. Recent developments in modeling languages based on advanced search algorithms advance both the logic programming and imperative programming sides. Modern Prolog implementations have added some of these advances, e.g., support for constraint programming and a module system.

- *Modeling languages (e.g., Comet, Numerica [48])*: The first paradigm is a solver: constraint programming (see Section 7), local search (see the chapter by Philippe Codognet [8]), satisfiability (SAT solvers), and so forth. The second paradigm is object-oriented programming.

- *Solving libraries (e.g., Gecode)*: The first paradigm is a solver library based on advanced search algorithms, such as Gecode [43, 47]. The second paradigm is added by the host language, e.g., C++ and Java support object-oriented programming.

- *Language embedding (e.g., SQL)*: SQL already supports two paradigms: a relational programming engine for logical queries of a database and a transactional interface for concurrent updates of the database. The host language complements this by supporting object-oriented programming, for organization of large programs. This example goes beyond two paradigms to show a design with three complementary paradigms.

3.2 A definitive programming language

At some point in time, language research will give solutions that are good enough that researchers will move on to work at higher levels of abstraction. This has already arrived for many subareas of language design, such as assembly languages and parsing algorithms. In the 1970s, compiler courses were built around a study of parsing algorithms. Today, parsing is well understood for most practical purposes and compiler design has moved on. Today's compiler courses are built around higher level topics such as dataflow analysis, type systems, and language concepts. We postulate that this kind of evolution is happening with language design as well.

Layer	Language project			
	Erlang [6, 5]	E [32, 31]	Distrib. Oz [10]	Didactic Oz [50]
Functional programming (see Section 4.2)	A process is a recursive function in its own thread, employing closures for hot code update	An object is a recursive function with a local state	Functions, procedures, classes, and components are closures with efficient distrib. protocols	Closures are the foundation of all paradigms
Deterministic concurrency (see Section 6)	(not supported)	Deterministic execution of all objects in one vat (process)	Dataflow concurrency with efficient protocol for dataflow variables	Concurrency is as easy as functional programming, no race conditions
Message-passing concurrency (see Section 4.3)	Fault tolerance by isolation, fault detection with messages	Security by isolation, messages between objects in different vats	Asynchronous message protocols to hide latency	Multi-agent programming is expressive and easy to program
Shared-state concurrency (see Section 4.4)	Global database (Mnesia) keeps consistent states	(not supported)	Coherent global state protocols; transactions for latency and fault tolerance	Named state for modularity

Table 1. Layered structure of a definitive programming language

This section presents the structure of one possible definitive language [52]. We study four research projects that were undertaken to solve four very different problems. The solutions achieved by all four projects are significant contributions to their respective areas. All four projects considered language design as a key factor to achieve success. The surprise is that all four projects ended up using languages with very similar structures. Table 1 shows the common properties of the programming language invented in each of the four projects. The common language has a layered structure with four layers: a strict functional core, followed by declarative concurrency, then asynchronous message passing, and finally global named state. This layered structure naturally supports four paradigms. We briefly summarize the four projects:

1. *Erlang* Programming highly available embedded systems for telecommunications. This project was undertaken by Joe Armstrong and his colleagues at the Ericsson Computer Science Laboratory starting in 1986. The Erlang language was designed and a first efficient and stable implementation was completed in 1991 [5, 6]. An Erlang program consists of isolated named lightweight processes that send each other messages. Because of the isolation, Erlang programs can be run almost unchanged on distributed systems and multi-core processors. The Erlang system has a replicated database, Mnesia, to keep global coherent states. Erlang and its programming platform, the OTP (Open Telecom Platform) system, are being used successfully in commercial systems by Ericsson and other companies [57, 17].

2. *E* Programming secure distributed systems with multiple users and multiple security domains. This project was undertaken over many years by different institutions. It started with Dennis and Van Horn's capability model in 1965 [13] and Carl Hewitt's Actor model in 1973 [24] and it led via concurrent logic programming to the E language designed by Doug Barnes, Mark Miller, and their colleagues [32, 31]. Predecessors of E have been used to implement various multiuser virtual

environments. An E program consists of isolated single-threaded vats (processes) hosting active objects that send each other messages. Deterministic concurrency is important in E because nondeterminism can support a covert channel.

3. *Distributed Oz* Making network-transparent distributed programming practical. This project started in 1995 in the PERDIO project at the DFKI with the realization that the well-factored design of the Oz language, first developed by Gert Smolka and his students in 1991 as an outgrowth of the ACCLAIM project, was a good starting point for making network transparent distribution practical [45]. This resulted in the Mozart Programming System which implements Distributed Oz and was first released in 1999 [22, 34]. Recent work has both simplified Mozart and increased its power for building fault-tolerance abstractions [10].

4. *Didactic Oz* Teaching programming as a unified discipline covering all popular programming paradigms. This project started in 1999 with the realization by the author and Seif Haridi that Oz is well-suited to teaching programming because it has many programming concepts in a well-factored design, it has a simple semantics, and it has a high-quality implementation. The textbook [50], published in 2004, "reconstructs" the Oz design according to a principled approach (see Section 2.3). The book is the basis of programming courses now being taught at several dozen universities worldwide. The author has been using it at UCL since 2003 for his second-year programming course given to all engineering students and his third-year concurrent programming course. The second-year course (since 2005 called FSAB1402) is particularly interesting since it covers the three most important paradigms, functional, object-oriented, and dataflow concurrent programming, with many practical techniques and a formal semantics [51].

From the common structure of these designs, one can infer several plausible consequences for language design. First, that the notion of declarative programming is at the very core of programming languages. This is already well-known; our study reinforces this conclusion. Second, that declarative programming will stay at the core for the foreseeable future, because distributed, secure, and fault-tolerant programming are essential topics that need support from the programming language. Third, that deterministic concurrency is an important form of concurrent programming that should not be ignored. We remark that deterministic concurrency is an excellent way to exploit the parallelism of multi-core processors because it is as easy as functional programming and it cannot have race conditions (see also Section 6) [53]. A final conclusion is that message-passing concurrency is the correct default for general-purpose concurrent programming instead of shared-state concurrency.

3.3 Architecture of self-sufficient systems

We have presented some preliminary conclusions about a definitive language; let us now be ambitious and widen our scope to software systems. The ultimate software system is one that does not require any human assistance, i.e., it can provide for every software modification that it needs, including maintenance, error detection and correction, and adaptation to changing requirements. Such a system can be called *self sufficient* [44]. Self-sufficient systems can be very robust; for example peer-to-peer networks can manage

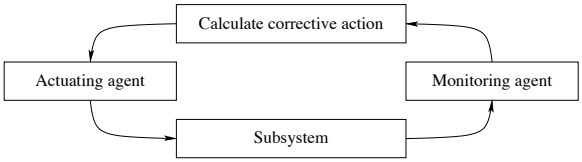

Figure 6. A single feedback loop

themselves to survive in extremely hostile environments by doing reversible phase transitions [44, 54]. Let us leave aside for now the artificial intelligence required to build such a system, and investigate just the language mechanisms it needs. The system may ask for human assistance in some cases, but in principle it should contain all the mechanisms it needs to accomplish its tasks.

What is a reasonable architecture for designing self-sufficient systems? From the conclusions of the previous section and our experience in building distributed systems, we can propose an architecture. In terms of programming paradigms, what we need first is components as first-class entities (specified by closures) that can be manipulated through higher-order programming. Above this level, the components behave as isolated concurrent agents that communicate through message passing. Finally, we need named state and transactions for system reconfiguration and system maintenance. Named state allows us to manage the content of components and change their interconnections. This gives us a language that has a layered structure similar to the previous section.

With this language we can program our system. To allow the program to adapt itself to its environment, we take inspiration from biological systems and organize its components as feedback loops. The system then consists of a set of interacting feedback loops. A single feedback loop consists of three concurrent components that interact with a subsystem (see Figure 6): a monitoring agent, a correcting agent, and an actuating agent. Realistic systems consist of many feedback loops. Since each subsystem must be as self-sufficient as possible, there must be feedback loops at all levels. These feedback loops can interact in two fundamental ways:

- Stigmergy: Two loops share one subsystem.
- Management: One loop controls another loop directly.

Figure 8 gives a real-world example from biology: the human respiratory system [49]. This system contains four loops. Three loops form a tower connected by management. The fourth loop interacts with the others through stigmergy.

The style of system design illustrated by the human respiratory system can be applied to programming. A program then consists of a set of feedback loops interacting through stigmergy and management. Figure 7 shows part of the Transmission Control Protocol as a feedback loop structure [49]. The inner loop implements reliable transfer of a byte stream using a sliding window protocol. The outer loop does congestion control: if too many packets are lost, it reduces the transfer rate of the inner loop by reducing the window size. In our view, the large-scale structure of software will more and more be done in this self-sufficient style. If it is not done in this way, the software will simply be too fragile and collapse with any random error or problem.

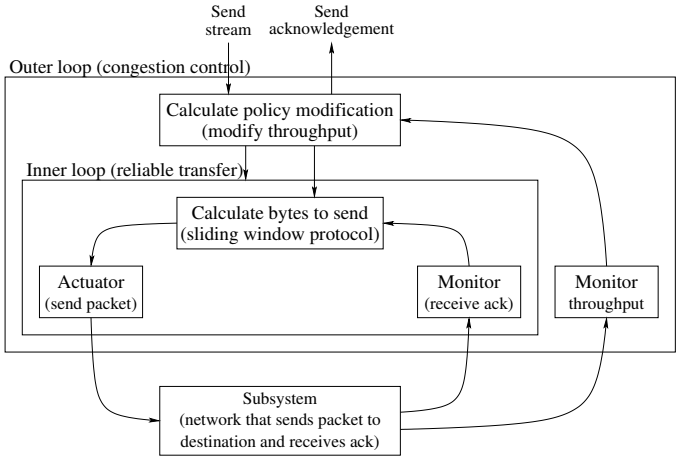

Figure 7. TCP as a feedback loop structure

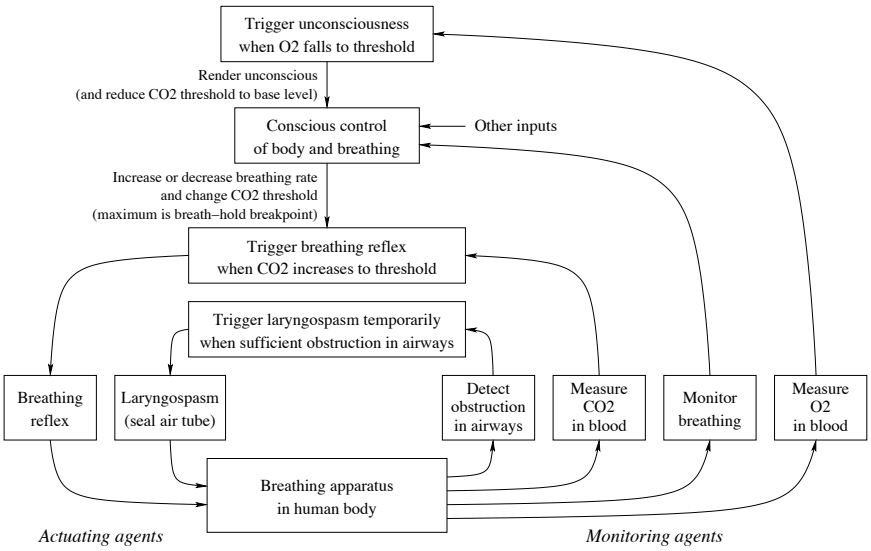

Figure 8. The human respiratory system as a feedback loop structure

4 Programming concepts

Programming paradigms are built out of programming concepts. In this section we present the four most important programming concepts, namely records, lexically scoped closures, independence (concurrency), and named state. We explain the concepts and why they are important for programming.

4.1 Record

A record is a data structure: a group of references to data items with indexed access to each item. For example:

```
R=chanson(nom:"Le Roi des Aulnes"
          artiste:"Dietrich Fischer-Dieskau"
          compositeur:"Franz Schubert"
          langue:allemand)
```

The record is referenced by the identifier `R`. Members can be references through the dot operation, e.g., `R.nom` returns a reference to the string `"Le Roi des Aulnes"`. The record is the foundation of symbolic programming. A symbolic programming language is able to calculate with records: create new records, decompose them, and examine them. Many important data structures such as arrays, lists, strings, trees, and hash tables can be derived from records. When combined with closures (see next section), records can be used for component-based programming.

4.2 Lexically scoped closure

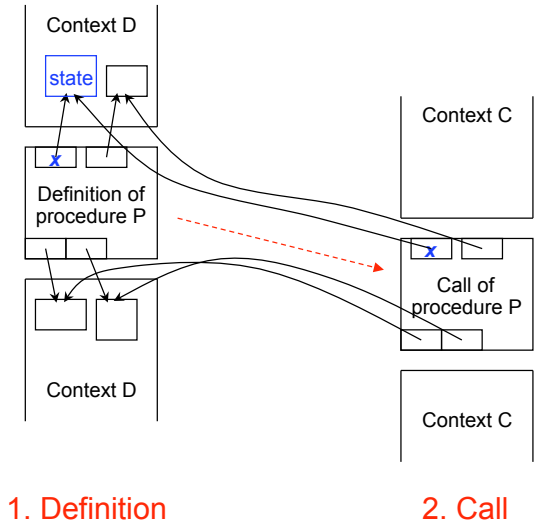

Figure 9. Definition and call of a closure

The lexically scoped closure is an enormously powerful concept that is at the heart of programming. Functional programming, which is programming with closures, is a

central paradigm (see Figure 2). From an implementation viewpoint, a closure combines a procedure with its external references (the references it uses at its definition). From the programmer's viewpoint, a closure is a "packet of work": a program can transform any instructions into a closure at one point in the program, pass it to another point, and decide to execute it at that point. The result of its execution is the same as if the instructions were executed at the point the closure was created.

Figure 9 shows schematically what happens when a closure is defined and when it is called. The procedure P is implemented by a closure. At the definition (context D), P stores the references from the definition context. For example, it keeps the reference x to some named state. We say that the environment (set of references) of P is *closed* over its definition context. At the call (context C), P uses the references from context D.

Figure 10 shows one possible use for a closure: creating a control structure. At the left, we execute the instruction <stmt>. At the right, instead of executing <stmt>, we place it inside a procedure (closure) referenced by P (the example uses Oz syntax). Any time later on in the program, we can decide to call P. We have separated the definition of <stmt> from its execution. With this ability we can define control structures such as an `if` statement or `while` loop.

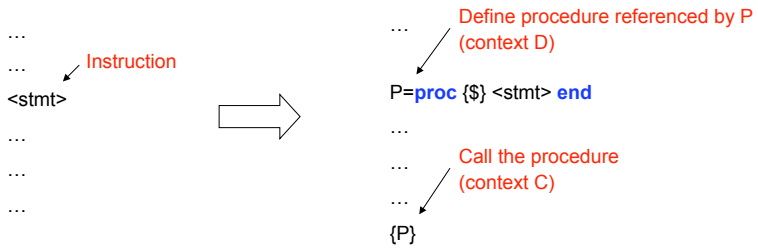

Figure 10. Example: modifying a program to separate creation and execution

The example of Figures 9 and 10 can easily be generalized to procedures with arguments. The closed environment exists as before. The arguments are passed during each call. The closure therefore has two sets of references: a closed environment (from the definition) and the arguments (from each call). Almost all programming languages (except for a few venerable ancestors such as Pascal and C) use this kind of closure:

- functions are closures;
- procedures are closures;
- objects are closures;
- classes are closures;
- software components are closures.

Many abilities normally associated with specific paradigms are based on closures:

- Instantiation and genericity, normally associated with object-oriented programming, can be done easily by writing functions that return other functions. In object-oriented programming the first function is called a "class" and the second is called an "object".

- Separation of concerns, normally associated with aspect-oriented programming, can be done easily by writing functions that take other functions as arguments. For example, Erlang has a function that implements a generic fault-tolerant client/server. It is called with a function argument that defines the server's behavior. Aspect-oriented programming in object-oriented languages is explained in the chapter by Pierre Cointe [9]. It is usually done by syntactic transformations (called "weaving") that add aspect code to the original source. The AspectJ language is a good example of this approach. Weaving is difficult to use because it is fragile: it is easy to introduce errors in the program (changing the source code changes the semantics of the program). Using closures instead makes it easier to preserve correctness because the source code is not changed.

- Component-based programming is a style of programming in which programs are organized as components, where each component may depend on other components. A *component* is a building block that specifies part of a program. An instance of a component is called a *module*, which is a record containing closures. A new module is created by a function that takes its dependent modules as inputs. The component is the function.

The Erlang language implements all these abilities directly with closures. This is practical and scalable: successful commercial products with more than one million lines of Erlang code have been developed (e.g., the AXD-301 ATM switch [57]). In most other languages, though, the use of closures is hidden inside the language's implementation and is not available directly to the programmer. If done carefully this can be an advantage, since the implementation can guarantee that the closures are used correctly.

4.3 Independence (concurrency)

Another key concept is independence: constructing a program as independent parts. This is not as simple as it may seem. For example, consider a program that consists of instructions executing one after the other. The instructions are not independent since they are ordered in time. To implement independence we need a new programming concept called concurrency. When two parts do not interact at all, we say they are *concurrent*.[3] (When the order of execution of two parts is given, we say they are *sequential*.) Concurrent parts can be extended to have some well-defined interaction, which is called communication.

Concurrency should not be confused with parallelism. Concurrency is a language concept and parallelism is a hardware concept. Two parts are parallel if they execute simultaneously on multiple processors. Concurrency and parallelism are orthogonal: it is possible to run concurrent programs on a single processor (using preemptive scheduling and time slices) and to run sequential programs on multiple processors (by parallelizing the calculations). Parallel execution on multi-core processors is explained on page 38.

The real world is concurrent: it consists of activities that evolve independently. The computing world is concurrent as well. It has three levels of concurrency:

[3] Technically, a program's execution consists of a partial order of state transition events and two events are *concurrent* if there is no order between them.

- Distributed system: a set of computers connected through a network. A concurrent activity is called a computer. This is the basic structure of the Internet.
- Operating system: the software that manages a computer. A concurrent activity is called a process. Processes have independent memories. The operating system handles the task of mapping the process execution and memory to the computer. For example, each running application typically executes in one process.
- Activities inside one process. A concurrent activity is called a thread. Threads execute independently but share the same memory space. For example, the different windows in a Web browser typically execute in separate threads.

The fundamental difference between processes and threads is how resource allocation is done. Process-level concurrency is sometimes called *competitive concurrency*: each process tries to acquire all the system's resources for itself. The operating system's chief role is to arbitrate the resource requests done by all the processes and to allocate resources in a fair way. Thread-level concurrency is sometimes called *cooperative concurrency*: threads in a process share resources and collaborate to achieve the result of the process. Threads run in the same application and so are guided by the same program.

There are two popular paradigms for concurrency. The first is *shared-state concurrency*: threads access shared data items using special control structures called monitors to manage concurrent access. This paradigm is by far the most popular. It used by almost all mainstream languages, such as Java and C#. Another way to do shared-state concurrency is by means of transactions: threads atomically update shared data items. This approach is used by databases and by software transactional memory. The second paradigm is *message-passing concurrency*: concurrent agents each running in a single thread that send each other messages. The languages CSP (Communicating Sequential Processes) [25] and Erlang [6] use message passing. CSP processes send synchronous messages (the sending process waits until the receiving process has taken the message) and Erlang processes send asynchronous messages (the sending process does not wait).

Despite their popularity, monitors are the most difficult concurrency primitive to program with [29]. Transactions and message passing are easier, but still difficult. All three approaches suffer from their expressiveness: they can express nondeterministic programs (whose execution is not completely determined by their specifications), which is why it is hard to reason about their correctness. Concurrent programming would be much simpler if the nondeterminism were controlled in some way, so that it is not visible to the programmer. Sections 6 and 7 present four important paradigms that implement this idea to make concurrent programming much simpler.

4.4 Named state

The final key concept we will introduce is named state. State introduces an abstract notion of time in programs. In functional programs, there is no notion of time. Functions are mathematical functions: when called with the same arguments, they always give the same results. Functions do not change. In the real world, things are different. There are few real-world entities that have the timeless behavior of functions. Organisms grow and learn. When the same stimulus is given to an organism at different times, the reaction will usually be different. How can we model this inside a program? We need to model an entity with a unique identity (its name) whose behavior changes during the execution

of the program. To do this, we add an abstract notion of time to the program. This abstract time is simply a *sequence of values in time* that has a *single name*. We call this sequence a named state. Unnamed state is also possible (monads and DCGs, see Section 2.1), but it does not have the modularity properties of named state.

Figure 11 shows two components, A and B, where component A has an internal named state (memory) and component B does not. Component B always has the same behavior: whenever it is called with the same arguments, it gives the same result. Component A can have different behaviors each time it is called, if it contains a different value in its named state. Having named state is both a blessing and a curse. It is a blessing because it allows the component to adapt to its environment. It can grow and learn. It is a curse because a component with named state can develop erratic behavior if the content of the named state is unknown or incorrect. A component without named state, once proved correct, always stays correct. Correctness is not so simple to maintain for a component with named state. A good rule is that named state should never be invisible: there should always be some way to access it from the outside.

Figure 11. A component with named state and a component without named state

Named state and modularity

Named state is important for a system's modularity. We say that a system (function, procedure, component, etc.) is *modular* if updates can be done to part of the system without changing the rest of the system. We give a scenario to show how we can design a modular system by using named state. Without named state, this is not possible.

Assume that we have three developers, P, U1, and U2. P has developed a module `M` that contains two functions `F` and `G`. U1 and U2 are users of `M`: their own programs used module `M`. Here is one possible definition of `M`:

```
fun {ModuleMaker}
   fun {F ...}
      ...  % Definition of F
   end
   fun {G ...}
      ...  % Definition of G
   end
in
   themodule(f:F g:G)
end
M={ModuleMaker}  % Creation of M
```

The function `ModuleMaker` is a software component, i.e., it defines the behavior of part of a system. We create instances of this component by calling `ModuleMaker`. One such instance is the module `M`. Note that a module's interface is simply a record, where each field is one of the module's operations. The module `M` has two operations `F` and `G`.

Now assume that developer U2 has an application that consumes a huge amount of calculation time. U2 would like to investigate where all this time is being spent, so that he can rewrite his application to be less costly. U2 suspects that `F` is being called too many times and he would like to verify this. U2 would like a new version of `M` that counts the number of times `F` is called. So U2 contacts P and asks him to create a new version of `M` that does this, but without changing the interface (that defines the operations of `M` and how they are called) since otherwise U2 would have to change all of his program (not to mention U1!).

Surprise! This is not possible without named state. If `F` does not have named state then it cannot change its behavior. In particular, it cannot keep a counter of how many times it is called. The only solution in a program without named state is to change `F`'s interface (its arguments):

```
fun {F ... Fin Fout}
   Fout=Fin+1
   ...
end
```

We add two arguments to `F`, namely `Fin` and `Fout`. When calling `F`, `Fin` gives the count of how many times `F` was called, and `F` calculates the new count in `Fout` by adding one to `Fin`. When calling `F`, we have to link all these new arguments together. For example, three successive calls to `F` would look like this:

```
A={F ... F1 F2}
B={F ... F2 F3}
C={F ... F3 F4}
```

`F1` is the initial count. The first call calculates `F2`, which is passed to the second call, and so forth. The final call returns the count `F4`. We see that this is a very bad solution, since U2 has to change his program wherever `F` is called. It gets worse: U1 also has to change his program, even though U1 never asked for any change. All users of `M`, even U1, have to change their programs, and they are very unhappy for this extra bureaucratic overhead.

The solution to this problem is to use named state. We give an internal memory to the module `M`. In Oz, this internal memory is called a *cell* or a *variable cell*. This corresponds simply to what many languages call a variable. Here is the solution:

```
fun {ModuleMaker}
   X={NewCell 0}  % Create cell referenced by X
   fun {F ...}
      X:=@X+1     % New content of X is old plus 1
      ...         % Original definition of F
   end
   fun {F ...}
      ...         % Original definition of G
   end
   fun {Count} @X end  % Return content of X
in
   themodule(f:F g:G c:Count)
end
M={ModuleMaker}
```

The new module `M` contains a cell inside. Whenever `F` is called, the cell is incremented. The additional operation `Count` (accessed by `M.c`) returns the current count of the cell. The interfaces of `F` and `G` are unchanged. Now everybody is happy: U2 has his new module and nobody has to change their programs at all since `F` and `G` are called in the same way. This shows how named state solves the problem of modularity.

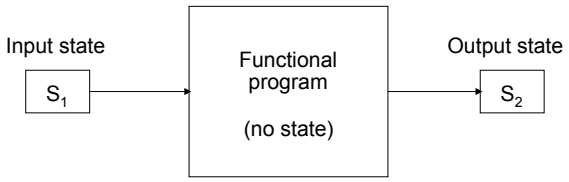

Figure 12. A program as state transformer

The main advantage of named state is that the program becomes modular. The main disadvantage is that a program can become incorrect. It seems that we need to have and not have named state at the same time. How do we solve this dilemma?[4] One solution is to concentrate the use of named state in one part of the program and to avoid named state in the rest. Figure 12 shows how this design works. The bulk of the program is a pure function without named state. The rest of the program is a state transformer: it calls the pure function to do the actual work. This concentrates the named state in a small part of the program.

5 Data abstraction

A data abstraction is a way to organize the use of data structures according to precise rules which guarantee that the data structures are used correctly. A data abstraction has an inside, an outside, and an interface between the two. All data structures are kept on the inside. The inside is hidden from the outside. All operations on the data must pass through the interface. Figure 13 shows this graphically. There are three advantages to this organization:

[4]This kind of dilemma is at the heart of invention. It is called a *technical contradiction* in Altshuller's Theory of Inventive Problem Solving (TRIZ), which provides techniques for its solution [2].

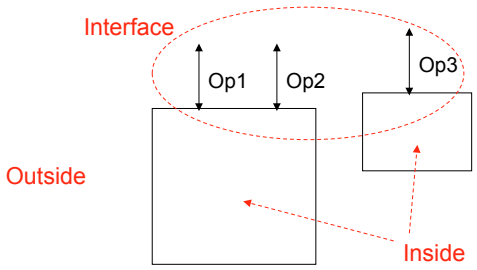

Figure 13. A data abstraction

1. First, there is a guarantee that the data abstraction will always work correctly. The interface defines the authorized operations on the data structures and no other operations are possible.

2. Second, the program is easier to understand. A user of the data abstraction does not need to understand how the abstraction is implemented. The program can be partitioned into many abstractions, implemented independently, which greatly reduces the program's complexity. This can be further improved by adding the property of compositionality: allowing data abstractions to be defined inside of other data abstractions.

3. Third, it becomes possible to develop very large programs. We can divide the implementation among a team of people. Each abstraction has one person who is responsible for it: he implements it and maintains it. That person has to know just the interfaces used by his abstraction.

In the rest of this section we first explain the four different ways to organize data abstractions. We then introduce two principles, polymorphism and inheritance, that greatly increase the power of data abstraction to organize programs. Object-oriented programming, as it is usually understood, is based on data abstraction with polymorphism and inheritance.

5.1 Objects and abstract data types

There are four main ways to organize data abstractions, organized along two axes. The first axis is *state*: does the abstraction use named state or not. The second axis is *bundling*: does the abstraction fuse data and operations into a single entity (this is called an *object* or a *procedural data abstraction (PDA)*), or does the abstraction keep them separate (this is called an *abstract data type (ADT)*). Multiplying the two axes gives four possibilities, which are shown in Figure 14.

Two of these four possibilities are especially popular in modern programming languages. We give examples of both in the Java language. Integers in Java are represented as values (1, 2, 3, etc.) and operations (+, -, *, etc.). The values are passed as arguments to the operations, which return new values. This is an example of an abstract data type without named state. Objects in Java combine the data (their attributes) and the operations (their methods) into a single entity. This is an example of an object with named state.

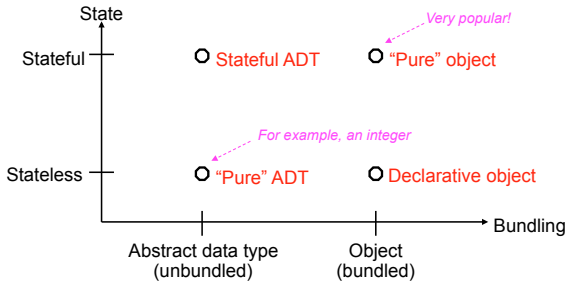

Figure 14. The four ways to organize a data abstraction

The two other possibilities, the abstract data type with named state and the declarative object, can also be useful. But they are less used in current languages.

5.2 Polymorphism and the responsability principle

The most important principle of object-oriented programming, after data abstraction itself, is polymorphism. In everyday language, we say an entity is polymorphic if it can take on different forms. In computer programming, we say an entity is polymorphic if it can take arguments of different types. This ability is very important for organizing large programs so that the responsibilities of the program's design are concentrated in well-defined places instead of being spread out over the whole program. To explain this, we use a real-world example. A sick patient goes to see a doctor. The patient does not need to be a doctor, but just to tell the doctor one message: "Cure me!". The doctor understands this message and does the right thing depending on his speciality. The program "GetCured" run by the patient is polymorphic: it takes a doctor as argument and works with all different kinds of doctors. This is because all doctors understand the message "Cure me!".

For programming the idea of polymorphism is similar: if a program works with one data abstraction as argument, it can work with another, *if* the other has the same interface. All four kinds of data abstractions we saw before support polymorphism. But it is particularly simple for objects, which is one reason for the success of object-oriented programming.

Figure 15 gives an example. Consider a graphics package which includes routines for drawing different kinds of figures. We define this using class declarations. Look at the definition of `CompoundFigure`. This defines figures that consist of a list of other figures (even other compound figures!). The method `draw` in `CompoundFigure` does not know how to draw any of these figures. But since it is polymorphic, it can call `draw` in the other figures. Each figure knows how to draw itself. This is a correct distribution of responsibilities.

```
class Figure
   ...
end

class Circle
   attr x y r
   meth draw ... end
   ...
end

class Line
   attr x1 y1 x2 y2
   meth draw ... end
   ...
end

class CompoundFigure
   attr figlist
   meth draw
      for F in @figlist do {F draw} end
   end
   ...
end
```

Figure 15. An example of polymorphism in a graphics package

5.3 Inheritance and the substitution principle

The second important principle of object-oriented programming is inheritance. Many abstractions have a lot in common, in what they do but also in their implementations. It can be a good idea to define abstractions to emphasize their common relationship and without repeating the code they share. Repeated code is a source of errors: if one copy is fixed, all copies have to be fixed. It is all too easy to forget some copies or to fix them in the wrong way.

Inheritance allows to define abstractions incrementally. Definition A can inherit from another definition B: definition A takes definition B as its base and shows how it is modified or extended. The incremental definition A is called a *class*. However, the abstraction that results is a full definition, not a partial one.

Inheritance can be a useful tool, but it should be used with care. The possibility of extending a definition B with inheritance can be seen as another interface to B. This interface needs to be maintained throughout the lifetime of B. This is an extra source of bugs. Our recommendation is to use inheritance as little as possible. When defining a class, we recommend to define it as nonextensible if at all possible. In Java this is called a `final` class.

Instead of inheritance, we recommend to use composition instead. Composition is a natural technique: it means simply that an attribute of an object refers to another object. The objects are composed together. In this way, it is not necessary to extend a class with inheritance. We use the objects as they are defined to be used. Figure 16 illustrates inheritance and composition side by side.

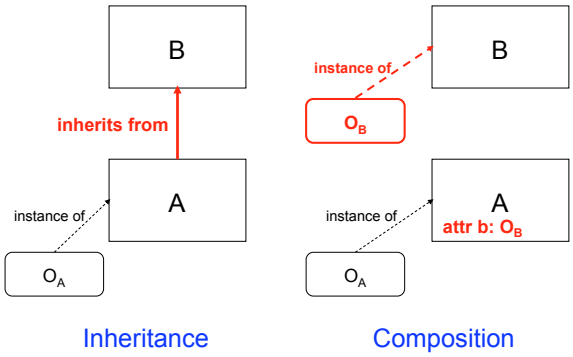

Figure 16. Inheritance versus composition

If you must use inheritance, then the right way to use it is to follow the substitution principle. Suppose that class A inherits from class B and we have two objects, O_A and O_B. The substitution principle states that any procedure that works with objects O_B of class B must also work with objects O_A of class A. In other words, inheritance should not break anything. Class A should be a conservative extension of class B.

We end our discussion of inheritance with a cautionary tale. In the 1980s, a very large multinational company[5] initiated an ambitious project based on object-oriented programming. Despite a budget of several billion dollars, the project failed miserably. One of the principal reasons for this failure was a wrong use of inheritance. Two main errors were committed:

- Violating the substitution principle. A procedure that worked with objects of a class no longer worked with objects of a subclass. As a result, many almost-identical procedures needed to be written.
- Using subclasses to mask bugs. Instead of correcting bugs, subclasses were created to mask bugs, i.e., to test for and handle those cases where the bugs occurred. As a result, the class hierarchy was very deep, complicated, slow, and filled with bugs.

6 Deterministic concurrent programming

One of the major problems of concurrent programming is nondeterminism. An execution of a program is nondeterministic if at some point during the execution there is a choice of what to do next. Nondeterminism appears naturally when there is concurrency: since two concurrent activities are independent, the program's specification cannot say which executes first. If there are several threads ready to run, then in each execution state the system has to choose which thread to execute next. This choice can be done in different ways; typically there is a part of the system called the *scheduler* that makes the choice.

Nondeterminism is very hard to handle if it can be observed by the user of the program. Observable nondeterminism is sometimes called a *race condition*. For example, if each of two threads assigns a variable cell to a different value, then each of the two values can be observed:

[5] Which shall remain anonymous.

```
declare C={NewCell 0}
thread C:=1 end
thread C:=2 end
```

The variable cell C can contain the value 1 or 2 after both threads execute. This is a simple case of a race condition. Much trickier cases are possible, when two threads share several variable cells and do more calculation with them. Debugging and reasoning about programs with race conditions is very difficult.

6.1 Avoiding nondeterminism in a concurrent language

The easiest way to eliminate race conditions is to design a language that does not have nondeterminism. But this would be throwing the baby out with the bathwater since concurrency naturally implies nondeterminism. How can we avoid the ill effects of nondeterminism and still have concurrency?[6] We can solve this problem by making a clear distinction between nondeterminism *inside* the system, which cannot be avoided, and *observable* nondeterminism, which may be avoidable. We solve the problem in two steps:

- First, we limit observable nondeterminism to those parts of the program that really need it. The other parts should have no observable nondeterminism.
- Second, we define the language so that it is possible to write concurrent programs without observable nondeterminism.

Concurrent paradigm	Races possible?	Inputs can be nondeterm.?	Example languages
Declarative concurrency	No	No	Oz [34], Alice [38]
Constraint programming	No	No	Gecode [43], Numerica [48]
Functional reactive programming	No	Yes	FrTime [12], Yampa [27]
Discrete synchronous programming	No	Yes	Esterel [7], Lustre [21], Signal [26]
Message-passing concurrency	Yes	Yes	Erlang [6], E [32]

Table 2. Four deterministic concurrent paradigms and one that is not

Is it possible to have a concurrent language without observable nondeterminism? A superficial examination of popular programming languages might lead one to say no: Java and C# use shared-state concurrency and Erlang uses message-passing concurrency, all of which have observable nondeterminism. Fortunately, this superficial impression is completely wrong. There are at least four useful programming paradigms that are concurrent but have no observable nondeterminism (no race conditions). Table 2 lists these four together with message-passing concurrency. Let us explain them in more detail.

[6] This is another example of a technical contradiction. See footnote on page 29.

Declarative concurrency (also called **monotonic dataflow)** In this paradigm, deterministic inputs are received and used to calculate deterministic outputs. This paradigm lives completely in a deterministic world. If there are multiple input streams, they must be deterministic, i.e., the program must know exactly what input elements to read to calculate each output (for example, there could be a convention that exactly one element is read from each input stream). Two languages that implement this paradigm are Oz [50, 34] and Alice [38]. This paradigm can be made lazy without losing its good properties. The paradigm and its lazy extension are explained in more detail in Section 6.2. Constraint programming is related to declarative concurrency and is explained in Section 7.

There exists also a *nonmonotonic* dataflow paradigm, in which changes on any input are immediately propagated through the program. The changes can be conceptualized as *dataflow tokens* traveling through the program. This paradigm can accept nondeterministic input, but it has the disadvantage that it sometimes adds its own nondeterminism that does not exist in the input (called a "glitch" below). That is why we do not discuss this paradigm further in this chapter. Functional reactive programming is similar to nonmonotonic dataflow but without the glitches.

Functional reactive programming (also called **continuous synchronous programming)** In this paradigm, programs are functional but the function arguments can be changed and the change is propagated to the output. This paradigm can accept nondeterministic input and does not add any nondeterminism of its own. Semantically, the arguments are continuous functions of a totally ordered variable (which can correspond to useful magnitudes such as *time* or *size*). Implementations typically recompute values only when they change and are needed. Discretization is introduced only when results are calculated [16]. This means that arbitrary scaling is possible without losing accuracy due to approximation. If the changes are propagated correctly, then the functional program does not add any nondeterminism. For example, the simple functional expression `x+(x*y)` with `x=3` and `y=4` gives `15`. If `x` is changed to `5`, then the expression's result changes from `15` to `25`. Implementing this naively with a concurrent stream connecting a times agent to a plus agent is incorrect. This implementation can give a *glitch*, for example if the new value of `x` reaches the addition before the new result of the multiplication. This gives a temporary result of `17`, which is incorrect. Glitches are a source of nondeterminism that the implementation must avoid, for example by compile-time preprocessing (doing a topological sort of operations) or thread scheduling constraints. Some languages that implement this paradigm are Yampa (embedded in Haskell) [27] and FrTime (embedded in Scheme) [12].

Discrete synchronous programming In this paradigm, a program waits for input events, does internal calculations, and emits output events. This is called a *reactive system*. Reactive systems must be deterministic: the same sequence of inputs produces the same sequence of outputs. Like functional reactive programming, this paradigm can accept nondeterministic input and does not add any nondeterminism of its own. The main difference is that time is discrete instead of continuous: time advances in steps from one input event to the next. Output events are emitted at the same logical time

instants as the input events.[7] All calculations done to determine the next output event are considered to be part of the same time instant. This is exactly what happens in clocked digital logic: combinational circuits are "instantaneous" (they happen within one cycle) and sequential circuits "take time": they use clocked memory (they happen over several cycles). The clock signal is a sequence of input events. Using discrete time enormously simplifies programming for reactive systems. For example, it means that subprograms can be trivially composed: output events from one subcomponent are instantaneously available as input events in other subcomponents. Some languages that implement this paradigm are Esterel [7], Lustre [21], and Signal [26]. Esterel is an imperative language, Lustre is a functional dataflow language, and Signal is a relational dataflow language. It is possible to combine the discrete synchronous and concurrent constraint paradigms to get the advantages of both. This gives the Timed CC model, which is explained in the chapter by Carlos Olarte *et al* [35].

All three paradigms have important practical applications and have been realized with languages that have good implementations. It is beyond the scope of this chapter to go into detail for all three paradigms. Because of its simplicity and importance, we give just the basic ideas of the first paradigm, declarative concurrency, in Section 6.2.

Deterministic concurrency and computer music

Deterministic concurrency is omnipresent in computer music. We give four examples:

- The OpenMusic music composition studio provides a set of graphical tools for composers [1]. It has an interactive visual language to define a dataflow graph of music components (called "patches"). It has a semantics similar to discrete synchronous programming. The main difference is explicit triggering. Explicit triggering is used to interface between the human composer and the system. The evaluation of a graph is triggered by the composer explicitly requesting the value of a component. This causes a demand-driven (lazy) chain of calculations: the component requests evaluation of the components that it depends on, and so forth transitively until reaching components that have no dependencies. The components have memory: the result of the evaluation is stored in the component.

- The Antescofo score follower lets music software follow human musicians when they are playing a piece [11]. It translates clock time (seconds) into human time (tempo, i.e., beats per minute). Antescofo is extended with a language that lets the composer annotate the score with control instructions. This language has a semantics similar to discrete synchronous programming, where input events are notes and output events are the composer's instructions. The main difference is that Antescofo has a tempo oscillator that can schedule events on fractional notes. This adds a continuous aspect to Antescofo's execution.

[7] Technically, a program in a synchronous language such as Esterel defines a deterministic *Mealy machine*, which is a finite state automaton in which each state transition is labeled with an input and an output.

- The Faust signal processing tool presented in the chapter by Yann Orlarey *et al* provides a visual dataflow language to define audio signal processing plug-ins or applications [37]. The dataflow language has a discrete synchronous semantics with a functional flavor, similar to Lustre [36]. Faust is optimized for high performance: it supports a high clock frequency and efficient compilation to C++.
- The Max/MSP programming and execution environment provides a set of graphical tools for music performance using an interactive visual language to define a dataflow graph [39]. Max/MSP has three distinct parts: the Max dataflow language, which provides the overall control flow, the MSP digital signal processing library, which generates the audio, and the Jitter library for video and 3D processing. The dataflow graph somewhat resembles that of OpenMusic, although the semantics is quite different. The Max language executes in real-time and in eager style starting from metronomes or other generators. The language is designed to make it easy to write functional dataflow programs (respecting the functional equations at all times, similar to functional reactive programming), although the implementation does not enforce this.[8] The language has a sequential semantics, since at most one message can be traversing the dataflow graph at any instant. The sequentiality is not immediately apparent to the user, but it is important for deterministic execution.

It is remarkable that these examples exist at three different levels of abstraction: music composition (OpenMusic), music performance (human time scale, Max/MSP and Antescofo), and music performance (signal processing time scale, Max/MSP and Faust).

OpenMusic, Max/MSP, and Antescofo provide a tantalizing confirmation of Table 1. OpenMusic has a mature language organized as three layers: functional, deterministic concurrency, and shared state. Max/MSP has a sequential core with a deterministic concurrent layer on top. Antescofo's language is still being designed: so far, it has just a deterministic concurrent layer, but other layers are planned. The Hermes/dl language, described in the chapter by Alexandre François, also distinguishes between deterministic and stateful layers [19].

6.2 Declarative concurrency

We explain briefly how to do declarative concurrency, which is the first and simplest form of deterministic concurrency (see chapter 4 of [50] for more information). Declarative concurrency has the main advantage of functional programming, namely confluence, in a concurrent model. This means that all evaluation orders give the same result, or in other words, it has no race conditions. It adds two concepts to the functional paradigm: threads and dataflow variables. A thread defines a sequence of instructions, executed independently of other threads. Threads have one operation:

- `{NewThread P}`: create a new thread that executes the 0-argument procedure `P`.

A dataflow variable is a single-assignment variable that is used for synchronization. Dataflow variables have three primitive operations:

[8] Strange errors sometimes appear at inconvenient times during the execution of Max/MSP programs. Some of these are likely due to program errors that result in temporary nondeterministic behavior, i.e., glitches. Such errors could be avoided by changing the language design or its run-time system, in ways similar to synchronous programming.

- `X={NewVar}`: create a new dataflow variable referenced by `X`.
- `{Bind X V}`: bind `X` to `V`, where `V` is a value or another dataflow variable.
- `{Wait X}`: the current thread waits until `X` is bound to a value.

Using these primitive operations, we extend all the operations of the language to wait until their arguments are available and to bind their result. For example, we define the operation `Add` in terms of dataflow variables and a primitive addition operation `PrimAdd`:

```
proc {Add X Y Z}
   {Wait X} {Wait Y}
   local R in {PrimAdd X Y R} {Bind Z R} end
end
```

The call `Z={Add 2 3}` causes `Z` to be bound to `5` (the function output is the procedure's third argument). We do the same for all operations including the conditional (**`if`**) statement (which waits until the condition is bound) and the procedure call (which waits until the procedure variable is bound). The result is a declarative dataflow language.

Lazy declarative concurrency

We can add lazy execution to declarative concurrency and still keep the good properties of confluence and determinism. In lazy execution, it is the consumer of a result that decides whether or not to perform a calculation, not the producer of the result. In a loop, the termination condition is in the consumer, not the producer. The producer can even be programmed as an infinite loop. Lazy execution does the least amount of calculation needed to get the result. We make declarative concurrency lazy by adding one concept, by-need synchronization, which is implemented by one operation:

- `{WaitNeeded X}`: the current thread waits until a thread does `{Wait X}`.

This paradigm adds both lazy evaluation and concurrency to functional programming and is still declarative. It is the most general declarative paradigm based on functional programming known so far.[9] With `WaitNeeded` we can define a lazy version of `Add`:

```
proc {LazyAdd X Y Z}
   thread {WaitNeeded Z} {Add X Y Z} end
end
```

This is practical if threads are efficient, such as in Mozart [34]. The call `Z={LazyAdd 2 3}` delays the addition until the value of `Z` is needed. We say that it creates a *lazy suspension*. If another thread executes `Z2={Add Z 4}`, then the suspension will be executed, binding `Z` to `5`. If the other thread executes `Z2={LazyAdd Z 4}` instead, then two lazy suspensions are created. If a third thread needs `Z2`, then both will be executed.

Declarative concurrency and multi-core processors

With the advent of multi-core processors, parallel programming has finally reached the mainstream. A multi-core processor combines two or more processing elements (called cores) in a single package, on a single die or multiple dies. The cores share the interconnect

[9] Constraint programming is more general but it is based on relational programming.

to the rest of the system and often share on-chip cache memory. As transistor density continues to increase according to Moore's Law (doubling approximately every two years, which is expected to continue at least until 2020) [33], the number of cores will increase as well. To use all this processing power we need to write parallel programs.

Decades of research show that parallel programming cannot be completely hidden from the programmer: it is not possible in general to automatically transform an arbitrary program into a parallel program. There is no magic bullet. The best that we can do is to make parallel programming as easy as possible. The programming language and its libraries should help and not hinder the programmer. Traditional languages such as Java or C++ are poorly equipped for this because shared-state concurrency is difficult.

Declarative concurrency is a good paradigm for parallel programming [53]. This is because it combines concurrency with the good properties of functional programming. Programs are mathematical functions: a correct function stays correct no matter how it is called (which is not true for objects). Programs have no race conditions: any part of a correct program can be executed concurrently without changing the results. Any correct program can be parallelized simply by executing its parts concurrently on different cores. If the set of instructions to execute is not totally ordered, then this can give a speedup. Paradigms that have named state (variable cells) make this harder because each variable cell imposes an order (its sequence of values). A common programming style is to have concurrent agents connected by streams. This kind of program can be parallelized simply by partitioning the agents over the cores, which gives a pipelined execution.

7 Constraint programming

In constraint programming, we express the problem to be solved as a constraint satisfaction problem (CSP). A CSP can be stated as follows: given a set of variables ranging over well-defined domains and a set of constraints (logical relations) on those variables, find an assignment of values to the variables that satisfies all the constraints. Constraint programming is the most declarative of all practical programming paradigms. The programmer specifies the result and the system searches for it. This use of search harnesses *blind chance* to find a solution: the system can find a solution that is completely unexpected by the programmer. The chapter by Philippe Codognet explains why this is useful for artistic invention [8].

Constraint programming is at a much higher level of abstraction than all the other paradigms in this chapter. This shows up in two ways. First, constraint programming can impose a *global condition* on a problem: a condition that is true for a solution. Second, constraint programming can actually *find* a solution in reasonable time, because it can use sophisticated algorithms for the implemented constraints and the search algorithm. This gives the solver a lot of power. For example, path-finding constraints can use shortest path algorithms, multiplication constraints can use prime factorization algorithms, and so forth. Because of its power in imposing both local and global conditions, constraint programming has been used in computer-aided composition [3, 41].

Programming with constraints is very different from programming in the other paradigms of this chapter. Instead of writing a set of instructions to be executed, the programmer *models* the problem: represent the problem using variables with their domains, define the problem as constraints on the variables, choose the propagators that implement the constraints, and define the distribution and search strategies. For small

constraint problems, a naive model works fine. For big problems, the model and heuristics have to be designed with care, to reduce search as much as possible by exploiting the problem structure and properties. The art of constraint programming consists in designing a model that makes big problems tractable.

The power and flexibility of a constraint programming system depend on the expressiveness of its variable domains, the expressiveness and pruning power of its propagators, and the smartness of its CSP solver. Early constraint systems were based on simple domains such as finite trees and integers. Modern constraint systems have added real numbers and recently also directed graphs as domains.

Constraint programming is closely related to declarative concurrency. Semantically, both are applications of Saraswat's concurrent constraint programming framework [42]. Like declarative concurrency, constraint programming is both concurrent and deterministic. It lives a deterministic world: for a given input it calculates a given output. It differs from declarative concurrency in two main ways. First, it replaces dataflow variables by general constraints. Binding a dataflow variable, e.g., `X=V`, can be seen as an equality constraint: `X` is equal to `V`. Second, it has a more flexible control flow: each constraint executes in its own thread, which makes it into a concurrent agent called a *propagator* (see Section 7.2). This allows the constraints to better prune the search space.

7.1 Some applications of constraint programming

Constraint programming has applications in many areas, such as combinatorics, planning, scheduling, optimization, and goal-oriented programming. The possible applications depend very much on the variable domains and constraints that are implemented in the solver. Simple combinatorial problems can be solved with integers. The variable domain corresponding to an integer is called a *finite domain* because it contains a finite set of integers. When we say, for example, that $x \in \{0, \cdots, 9\}$, we mean that the solution for x is an element of the finite set $\{0, \cdots, 9\}$. If we have eight variables s, e, n, d, m, o, r, y, all in the set $\{0, \cdots, 9\}$, then we can model the SEND+MORE=MONEY puzzle (where each letter represents a digit) with the single constraint $1000s+100e+10n+d+1000m+100o+10r+e = 10000m+1000o+100n+10e+y$. We add the constraints $s > 0$ and $m > 0$ to ensure the first digits are nonzero and the constraint *alldiff*$(\{s, e, n, d, m, o, r, y\})$ to ensure that all digits are different. To solve this problem intelligently, the constraint solver needs just one more piece of information: a heuristic known as the distribution strategy (see Section 7.2). For this example, a simple heuristic called first-fail is sufficient.

Finite domains are a simple example of a discrete domain. Constraint systems have also been built using continuous domains. For example, the Numerica system uses real intervals and can solve problems with differential equations [48]. The difference between Numerica's techniques and the usual numerical solution of differential equations (e.g., Runge-Kutta or predictor-corrector methods) is that the constraint solver gives a *guarantee*: the solution, if it exists is guaranteed to be in the interval calculated by the solver. The usual methods give no guarantee but only an approximate error bound.

Graph constraints and computer music

Recent research since 2006 has introduced a very powerful discrete domain, namely directed graphs. Variables range over directed graphs and the constraints define conditions

on the graphs. These can include simple conditions such as existence or nonexistence of edges or nodes. But what makes the domain truly interesting is that it can also include complex conditions such as transitive closure, the existence of paths and dominators, and subgraph isomorphisms [15, 40, 58]. The complex conditions are implemented by sophisticated graph algorithms. A Gecode library for graph constraints is in preparation as part of the MANCOOSI project [20, 43].

Graph constraints can be used in any problem where the solution involves graphs. The MANCOOSI project uses them to solve the package installability problem for large open-source software distributions. Spiessens has used graph constraints to reason about authority propagation in secure systems [46]. The nodes of an authority graph are subjects and objects. An edge in an authority graph describes a permission: an entity has a right to perform an action on another entity. A path in an authority graph describes an authority: an entity can perform an action, either directly or indirectly. Authority propagation problems can be formulated as graph problems. Since constraint programs are relational, this works in both directions: to find the use conditions for a system with given security properties or the security properties of a system with given use conditions.

A piece of music has a global order. A music score can be represented as a graph. Because of these two facts, we hypothesize that graph constraints can be useful primitives for computer-aided composition. For example, subgraph isomorphism can be used to find or to impose themes throughout a composition. Probably it will be necessary to design new graph constraints for computer music. For example, in a music score, the same theme can often be found in different places and at different time scales, perhaps giving the score a fractal structure. A global constraint can be designed to enforce this condition.

7.2 How the constraint solver works

In principle, solving a CSP is easy: just enumerate all possible values for all variables and test whether each enumeration is a solution. This naive approach is wildly impractical. Practical constraint solvers use much smarter techniques such as local search (explained in the chapter by Philippe Codognet [8]) or the propagate-distribute algorithm (explained in this section). The latter reduces the amount of search by alternating propagate and distribute steps (for more information see [47], which explains the Gecode library):

- *Propagate step*: Reduce the domains of the variables in size as much as possible according to the propagators. A *propagator* is a concurrent agent that implements a constraint. It is triggered when the domains of any of its arguments change. It then attempts to further reduce the domains of its arguments according to the constraint it implements. Propagators can trigger each other through shared arguments. They execute until no more reduction is possible (a fixpoint). This leads to three possibilities: a solution, a failure (no solution), or an incomplete solution.
- *Distribute step*: For each incomplete solution, choose a constraint C and split the problem P into two subproblems $P \wedge C$ and $P \wedge \neg C$. This increases the number of problems to solve, but each problem may be easier to solve since it has extra information (C or $\neg C$). This step is the most primitive form of search.

The algorithm then continues with propagate steps for the two subproblems. This creates a binary tree called the search tree. The efficiency of the propagate-distribute algorithm depends on three factors that can be chosen independently (see Figure 17):

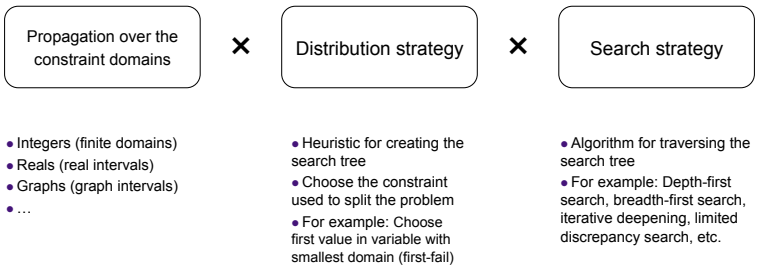

Figure 17. Constraint solver based on the propagate-distribute algorithm

- *Propagation over the constraint domains.* This defines how much propagation (pruning) is done by the propagators. This depends on two factors: the sophistication of the propagators and the expressiveness of the constraint domains. Propagators can implement highly sophisticated algorithms that depend on deep theoretical results. For example, the multiplication propagator `A*B=:C` can use factorization algorithms to improve propagation. For positive integers `A` and `B`, the multiplication propagator `A*B=:12` will either reduce to `A`,`B`$\in \{1, \cdots, 12\}$ or `A`,`B`$\in \{1, 2, 3, 4, 6, 12\}$, depending on whether the constraint domains can have "holes" or not. Better propagation and more expressive constraint domains reduce the number of distribution steps (less search) at the cost of more propagation (more inferencing). Depending on the problem, this may or may not be a good trade-off.
- *Distribution strategy.* This heuristic defines how the constraint C is chosen for each distribute step. A good choice of C depends on the structure of the problem and the distribution of its solutions. For example, the first-fail heuristic finds the variable with the smallest domain and chooses the first value in this domain.
- *Search strategy.* This heuristic defines how the search tree is traversed. Typical traversals are depth-first or breadth-first, but many more sophisticated traversals exist, such as A*, iterative deepening, and limited discrepancy. A* finds the shortest path by guiding the search with a heuristic function: actual distance traveled plus estimated remaining distance to goal. The estimation must not be greater than the actual remaining distance. Iterative deepening and limited discrepancy do progressively wider searches, starting with a bound of 1 and incrementing the bound after each traversal. Iterative deepening uses a depth bound and limited discrepancy uses a discrepancy bound (for example, the number of differences with respect to a depth-first path).

8 Conclusions and suggestions for going further

The chapter gives a quick overview of the main programming paradigms and their concepts. Programming languages should support several paradigms because different problems require different concepts to solve them. We showed several ways to achieve this: dual-paradigm languages that support two paradigms and a definitive language with four paradigms in a layered structure. Each paradigm has its own "soul" that can only be understand by actually using the paradigm. We recommend that you explore the paradigms

by actually programming in them. Each paradigm has programming languages that support it well with their communities and champions. For example, we recommend Haskell for lazy functional programming [28], Erlang for message-passing concurrency [6], SQL for transactional programming, Esterel for discrete synchronous programming [7], and Oz for declarative concurrency and constraint programming [50].

If you want to explore how to use different paradigms in one program, we recommend a multiparadigm language like Oz [34], Alice [38], Curry [4], or CIAO [23]. Each of these four has its own distinctive approach; pick the one you like best! For Oz there is a textbook and a Web site that has much material including full courses in English and French [50, 51]. There is a big difference between a language that is designed from the start to be multiparadigm (like Oz) and a language that contains many programming concepts (like Common Lisp). A true multiparadigm language is factored: it is possible to program in one paradigm without interference from the other paradigms.

Acknowledgements

This chapter was written during the author's sabbatical at IRCAM. Part of the work reported here is funded by the European Union in the SELFMAN project (sixth framework programme contract 34084) and the MANCOOSI project (seventh framework programme grant agreement 214898). The author thanks Arshia Cont, the developer of Antescofo, and Gérard Assayag and the other members of the RepMus group at IRCAM for interesting discussions that led to some of the conclusions in this chapter.

Bibliography

[1] Agon C., Assayag G. and Bresson B., *OpenMusic 6.0.6*, IRCAM, Music Representations Research Group, 2008.

[2] Altshuller G., "The Innovation Algorithm: TRIZ, Systematic Innovation and Technical Creativity," Technical Innovation Center, Inc. (Translated from the Russian by Lev Shulyak and Steven Rodman), 2005.

[3] Anders T., *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System*, Ph.D. dissertation, Queen's University, Belfast, Northern Ireland, Feb. 2007.

[4] Hanus A., Hanus S., and Hanus M., *Curry: A Tutorial Introduction*, Dec. 2007. See `www.curry-language.org`.

[5] Armstrong J., *Making Reliable Distributed Systems in the Presence of Software Errors*, Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, Nov. 2003.

[6] Armstrong J., Williams M., Wikström C. and Virding R., *Concurrent Programming in Erlang*, Prentice-Hall, 1996. See `www.erlang.org`.

[7] Berry G., *The Esterel v5 Language Primer–Version 5.21 release 2.0*, École des Mines and INRIA, April 1999.

[8] Codognet P. "Combinatorics, Randomness, and the Art of Inventing", *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[9] Cointe P. "Designing Open-Ended Languages: an Historical Perspective", *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[10] Collet R., *The Limits of Network Transparency in a Distributed Programming Language*, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, Dec. 2007.

[11] Cont A., "Antescofo: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music", *ICMC 2008*, Belfast, Northern Ireland, Aug. 2008.

[12] Cooper G. H., *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*, Ph.D. dissertation, Brown University, Providence, Rhode Island, May 2008.

[13] Dennis J.B. and Van Horn E.C., "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM*, 9(3), March 1966.

[14] Dijkstra E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[15] Dooms G., *The CP(Graph) Computation Domain in Constraint Programming*, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.

[16] Elliott C., *Simply Efficient Functional Reactivity*, LambdaPix technical report 2008-01, April 2008.

[17] Ericsson, *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts*, Telefonaktiebolaget LM Ericsson, Stockholm, Sweden, 1996.

[18] Felleisen M., "On the Expressive Power of Programming Languages", in *3rd European Symposium on Programming (ESOP 1990)*, May 1990, pp. 134-151.

[19] François, A. R. J., "Time and Perception in Music and Computation", *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[20] Gutiérrez G., *Graph Constraint Library for Gecode*, MANCOOSI project deliverable, see `www.mancoosi.org`. In preparation, 2009.

[21] Halbwachs N. and Pascal R., *A Tutorial of Lustre*, Jan. 2002.

[22] Seif H., Van Roy P., Brand P., and Schulte C., "Programming Languages for Distributed Applications", *Journal of New Generation Computing*, 16(3), pp. 223-261, May 1998.

[23] Hermenegildo M. V., Bueno F., Carro M., López P., Morales J. F., and Puebla G., "An Overview of the Ciao Multiparadigm Language and Program Development Environment and Its Design Philosophy", Springer LNCS 5065 (Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday), pp. 209-237, 2008.

[24] Hewitt C., Bishop P. and Steiger R., "A Universal Modular ACTOR Formalism for Artificial Intelligence". In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235-245, Aug. 1973.

[25] Hoare C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[26] Houssais B., *The Synchronous Programming Language SIGNAL: A Tutorial*, IRISA, April 2002.

[27] Hudak P., Courtney A., Nilsson H., and Peterson J., "Arrows, Robots, and Functional Reactive Programming". In *summer School on Advanced Functional programming*, Springer LNCS 2638, pp. 159-187, 2003.

[28] Hudak P., Peterson J. and Fasel J., *A Gentle Introduction to Haskell, Version 98*, See `www.haskell.org/tutorial`.

[29] Lea D., *Concurrent Programming in Java: Design Principles and Patterns*, Prentice Hall, 1999.

[30] Lienhard M., Schmitt A. and Stefani J.-B., *Oz/K: A Kernel Language for Component-Based Open Programming*. In *Sixth International Conference on Generative Programming and Component Engineering (GPCE'07)*, Oct. 2007.

[31] Miller M. S., "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, May 2006.

[32] Miller M S., Stiegler M., Close T., Frantz B., Yee K-P., Morningstar C., Shapiro J., Hardy N., Tribble E .D., Barnes D., Bornstien D., Wilcox-O'Hearn B., Stanley T., Reid K. and Bacon D., *E: Open Source Distributed Capabilities*, 2001.
See `www.erights.org`.

[33] Moore G. E., *Moore's Law*, Wikipedia, the free encyclopedia, 2009.

[34] Mozart Consortium, *Mozart Programming System*, Version 1.4.0 released July 2008. See `www.mozart-oz.org`.

[35] Olarte C., Rueda C. and Valencia F. D., "Concurrent Constraint Calculi: a Declarative Paradigm for Modeling Music Systems". *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[36] Orlarey Y., Fober D. and Letz S., "Syntactical and Semantical Aspects of Faust". *Soft Comput. 8(9)*, Nov. 2004, pp. 623-632.

[37] Orlarey Y., Fober D. and Letz S., "FAUST: an Efficient Functional Approach to DSP Programming". *New Computational Paradigms for Computer Music*, Assayag G. and Gerzso A. (eds.), IRCAM/Delatour France, 2009.

[38] Programming Systems Lab, Saarland University, *Alice ML Version 1.4*. See `www.ps.uni-sb.de/alice`.

[39] Puckette M., *et al.*, *Max/MSP Version 5.0.6*, 2009. See `www.cycling74.com`.

[40] Quesada L., *Solving Constraint Graph Problems using Reachability Constraints based on Transitive Closure and Dominators*, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, Nov. 2006.

[41] Rueda C., Alvarez G., Quesada L., Tamura G., Valencia F., Díaz J. F. and Assayag G., "Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language", *Constraints Journal*, Kluwer Academic Publishers, number 6, pp. 21-52, 2001.

[42] Saraswat, V. A. *Concurrent Constraint Programming*, MIT Press, Cambridge, MA, 1993.

[43] Schulte C., Lagerkvist M. and Tack G., *Gecode: Generic Constraint Development Environment*, 2006. See `www.gecode.org`.

[44] SELFMAN project. *Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components*, European Commission 6th Framework Programme, 2006-2009. See `www.ist-selfman.org`.

[45] Smolka G., Schulte C. and Van Roy P., *PERDIO–Persistent and Distributed Programming in Oz*, BMBF project proposal, DFKI, Saarbrücken, Germany. Feb. 1995.

[46] Spiessens A., *Patterns of Safe Collaboration*, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, Feb. 2007.

[47] Tack G., *Constraint Propagation: Models, Techniques, Implementation*, Ph.D. dissertation, Saarland University, Saarbrücken, Germany, Jan. 2009.

[48] Van Hentenryck P., "A Gentle Introduction to NUMERICA", *Artif. Intell.* 103 (1-2), Aug. 1998, pp. 209-235.

[49] Van Roy P., "Self Management and the Future of Software Design", *Third International Workshop on Formal Aspects of Component Software (FACS 2006)*, ENTCS volume 182, June 2007, pages 201-217.

[50] Van Roy P. and Seif H., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, Cambridge, MA, 2004. See `ctm.info.ucl.ac.be`.

[51] Van Roy P. and Seif H., *Programmation: Concepts, Techniques et Modèles* (in French), Dunod Éditeur, 2007. See `ctm.info.ucl.ac.be/fr`.

[52] Van Roy P., "Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place". In *8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, Fuji Sosono, Japan, Springer LNCS 3945, April 2006, pp. 2-12.

[53] Van Roy P., "The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard", Position statement, in *ICMC 2008*, Belfast, Northern Ireland, Aug. 2008.

[54] Van Roy P., " Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions", In *BCS Symposium on Visions of Computer Science*, London, UK, Sept. 2008.

[55] Van Roy P., *The Principal Programming Paradigms*, Poster version 1.08. See `www.info.ucl.ac.be/~pvr/paradigms.html`, 2008.

[56] Weinberg G., *An Introduction to General Systems Thinking*, Dorset House Publishing Co., 1977 (silver anniversary edition 2001).

[57] Wiger U., "Four-Fold Increase in Productivity and Quality–Industrial-Strength Runctional Programming in Telecom-Class Products", In *Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems*, 2001.

[58] Zampelli S., *A Constraint Programming Approach to Subgraph Isomorphism*, Ph.D. dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, June 2008.