Language Paradigms

Programming Language Design 2021

Hans Hüttel

March 2021

Part I

Introduction

What is a language paradigm?

The word paradigm comes from the Greek $\pi\alpha\rho\acute{\alpha}\delta\epsilon\iota\gamma\mu\alpha$ (pattern, example). It usually denotes a group of things or concepts that share significant properties. This is also the case for programming languages.

A paradigm is, hence, *a selection of properties* that a language can have or not have.

Learning goals

- To be able to classify programming languages according to paradigms based on data flow, execution ordering and structuring, respectively
- To understand and explain object-oriented languages that use simple inheritance, multiple inheritance and prototypes, respectively, and understand and explain the pointer models underlying these three notions
- To be able to understand and use the terminology underlying the syntax of Prolog
- To be able to read and understand simple programs in pure Prolog
- To be able to understand and explain the underlying ideas of resolution and how they are used in logic programming, including the notions of unification and backtracking

How to classify programming languages

We can classify programming languages according to different kinds of properties.

- Data flow: How are values carried through a program?
- Execution order: How are different parts of the execution ordered in time?
- Structuring: How are programs structured?

Additionally, we will take a closer look at some specific paradigms.

Data flow

Imperative: Data is communicated by modifying the contents of *locations* (variables, memory) that are shared by different parts of the program.

Functional/applicative: Data is communicated by giving values as arguments to function calls and by returning values as results from function calls.

Logical: Data is communicated through *logical variables*: shared locations that are initially uninitialised, but may be initialised by any of the program parts that share these locations, but not modified after initialisation (write-once, read-many) – except through backtracking.

Message passing: Data is transferred as messages from one process to another.

Execution order

Sequential. The syntax of the program indicates a well-defined (though possibly conditional) order in which execution/evaluation happens.

Parallel. Multiple operations are done at the same time, either by executing different parts of the program at the same time (thread parallelism) or by executing one piece of the program on multiple, different data at the same time (data parallelism).

Concurrent. Execution of different parts (called *processes* or *threads*) of the program is *interleaved*.

Declarative. Execution order is mostly unspecified, so a program specifies *what* should be done, but not in which order this is done. Dependencies through data may impose a partial order on computations. In contrast to concurrency, the choice of execution order does not affect the final result.

Structuring

Block structured languages allow explicitly delimited blocks of declarations and code to be combined by nesting and sequence.

Procedural languages allow procedures or functions to be declared so these can be called from other parts of a program.

Modular languages allow collections of declarations to be collected into named *modules* that can be used by multiple programs or other modules. A module may be required to implement a *signature* that specifies a set of names that must be declared in the module and required properties of these, such as their types.

Object-oriented languages structure programs using *objects*, where an object is a value that contains variables (called *fields*) and procedures (called *methods*) that can access the fields in the same object. An object can be an instance of a class or cloned from another object.

Part II

Object-oriented languages

Object-oriented languages

The object-oriented programming is (also) within the imperative paradigm.

It all began with the Norwegian SIMULA language in the 1960s due to Ole-Johan Dahl, Kristen Nygaard and others.

Another highly influential language was Smalltalk, developed at Xerox by Adele Goldberg and David Robson. The pointer semantics for classes is due to them.

Object-oriented languages

Object-oriented languages are often class-based.

An object is then usually implemented as a record of fields and a pointer to a *class descriptor* that points to implementations of methods.

There are also classless languages that use a notion of prototypes.

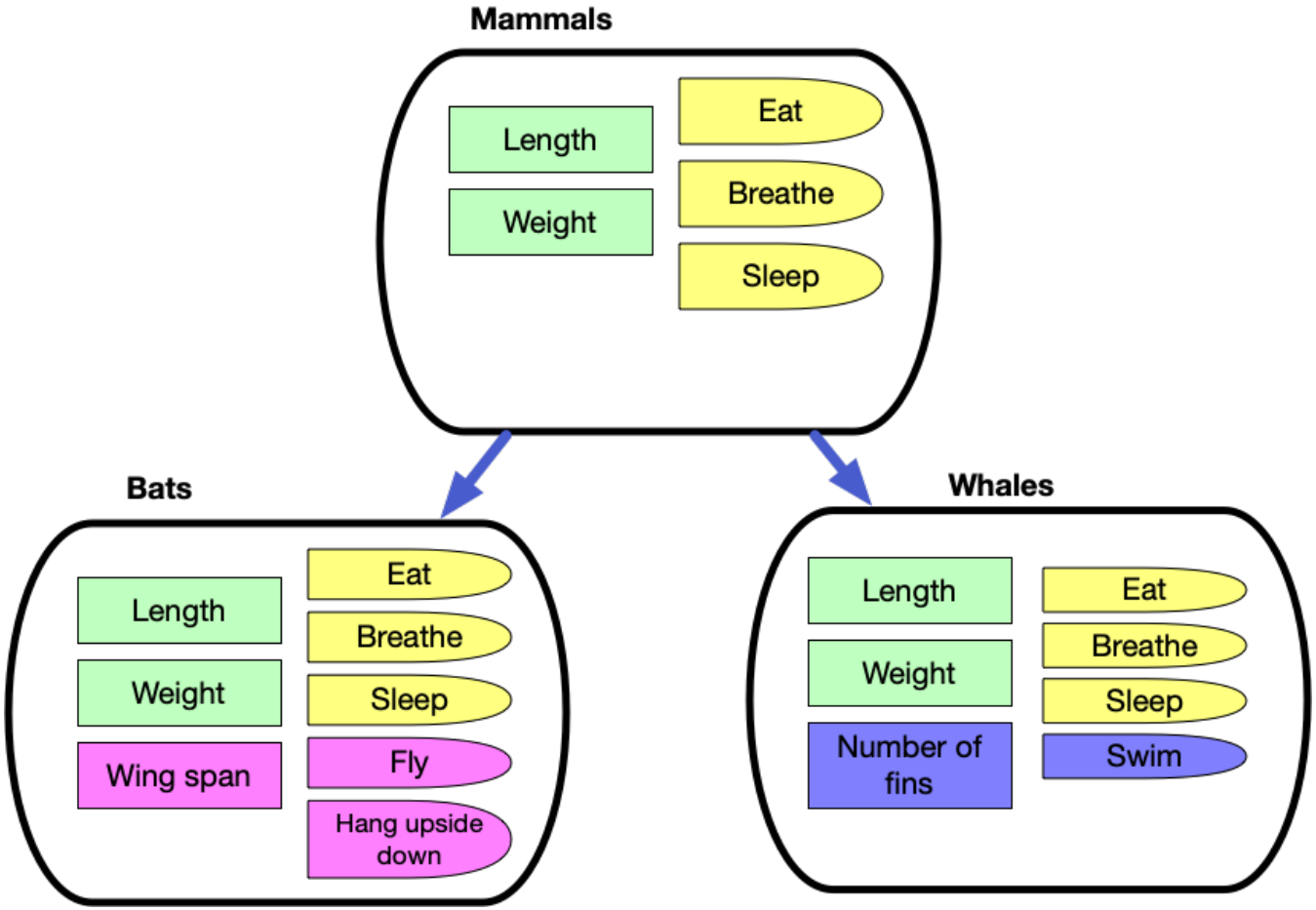
Different approaches to classifying objects

Single inheritance A class inherits from (at most) one superclass.

Multiple inheritance A class can inherit from several superclasses.
The class hierarchy is an acyclic graph.

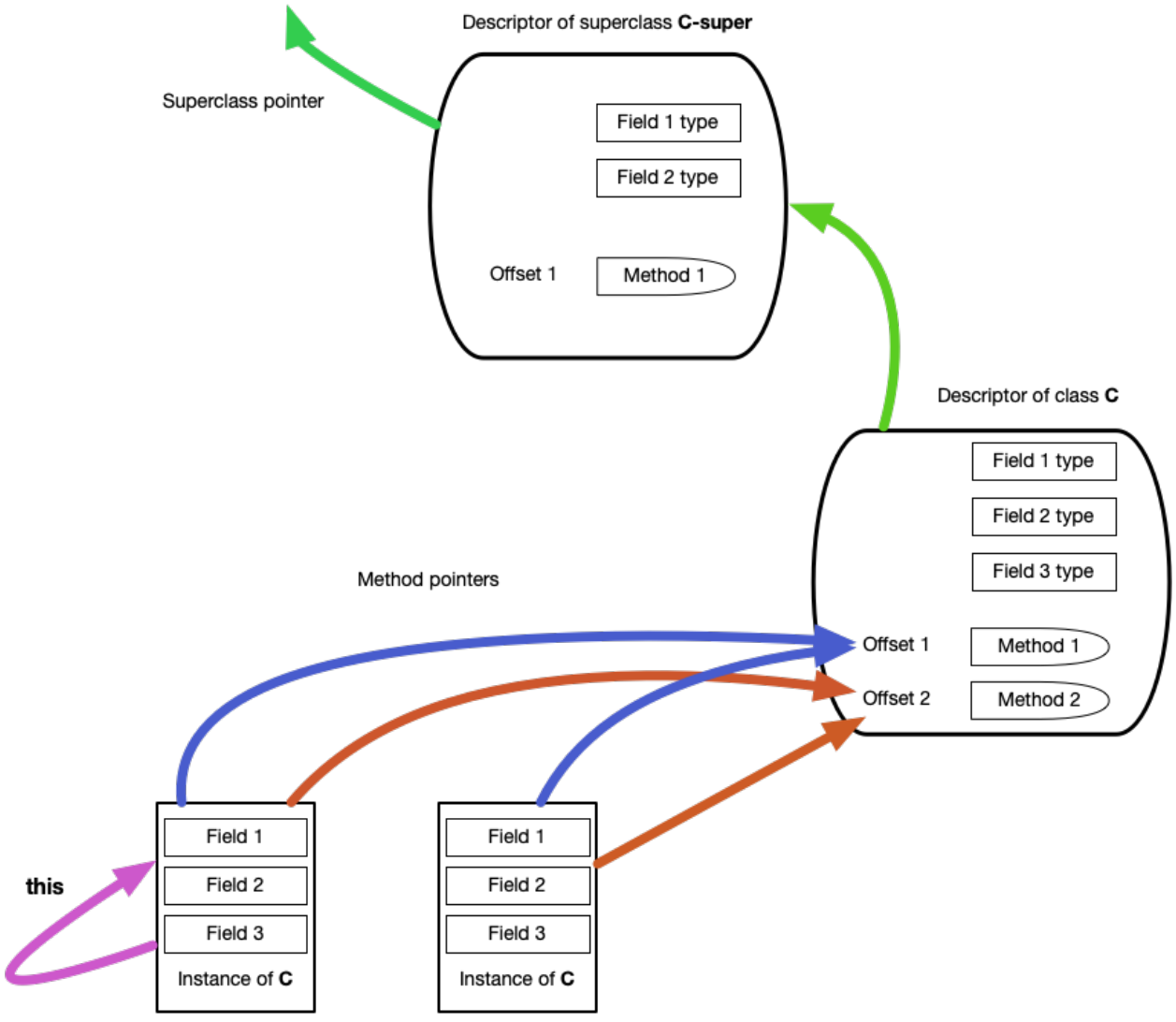
Prototype-based classless languages There is no notion of class!
New objects are created by cloning and extension.
We use dynamic interfaces aka duck typing.

Single inheritance



The class hierarchy is a tree.

Implementing single inheritance



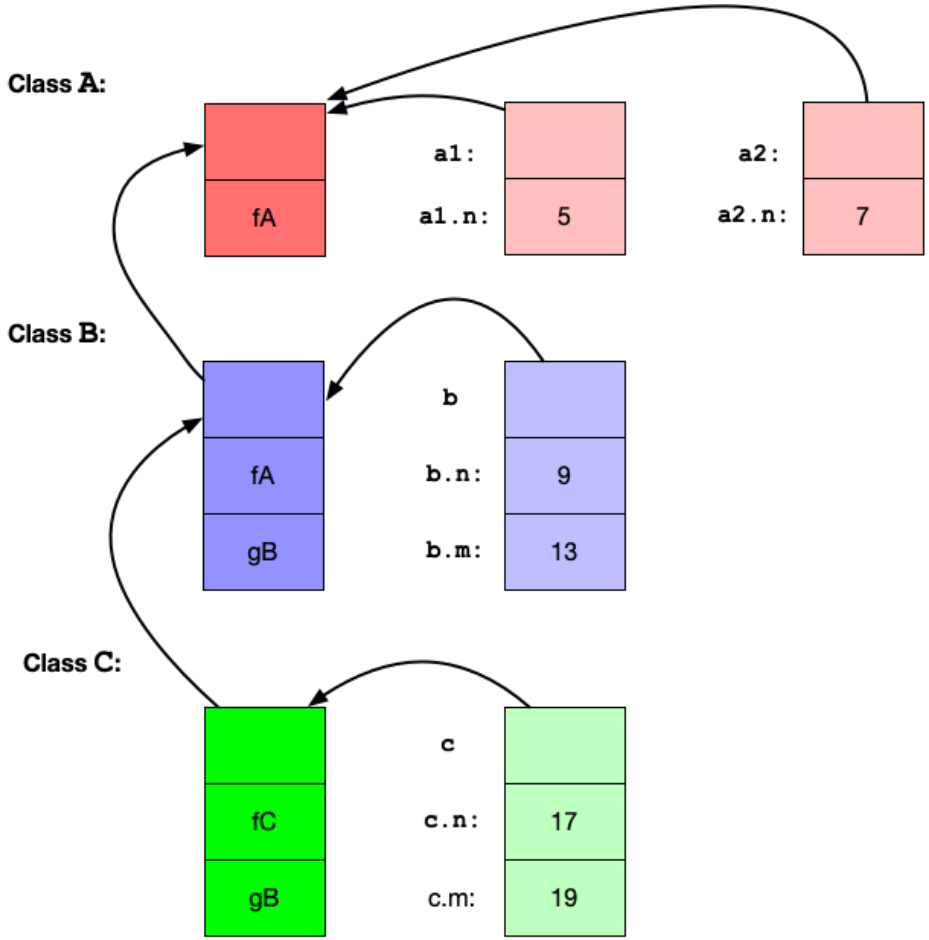
Implementing single inheritance

- An object is a record of its fields plus a pointer to a class descriptor. The fields have statically known offsets.
- A class descriptor is shared between all objects of a class. It has pointers to the methods of the class and a pointer to the descriptor of its superclass (if any). A method is identified by a offset.
- In a subclass, new methods and fields have offsets higher than those in the superclass, but inherited fields and methods have the same offsets – even if a method is overridden.
- When a method is called, an extra implicit "this" parameter is added to allow methods to access fields.
- Downcasts are checked by following superclass pointers.

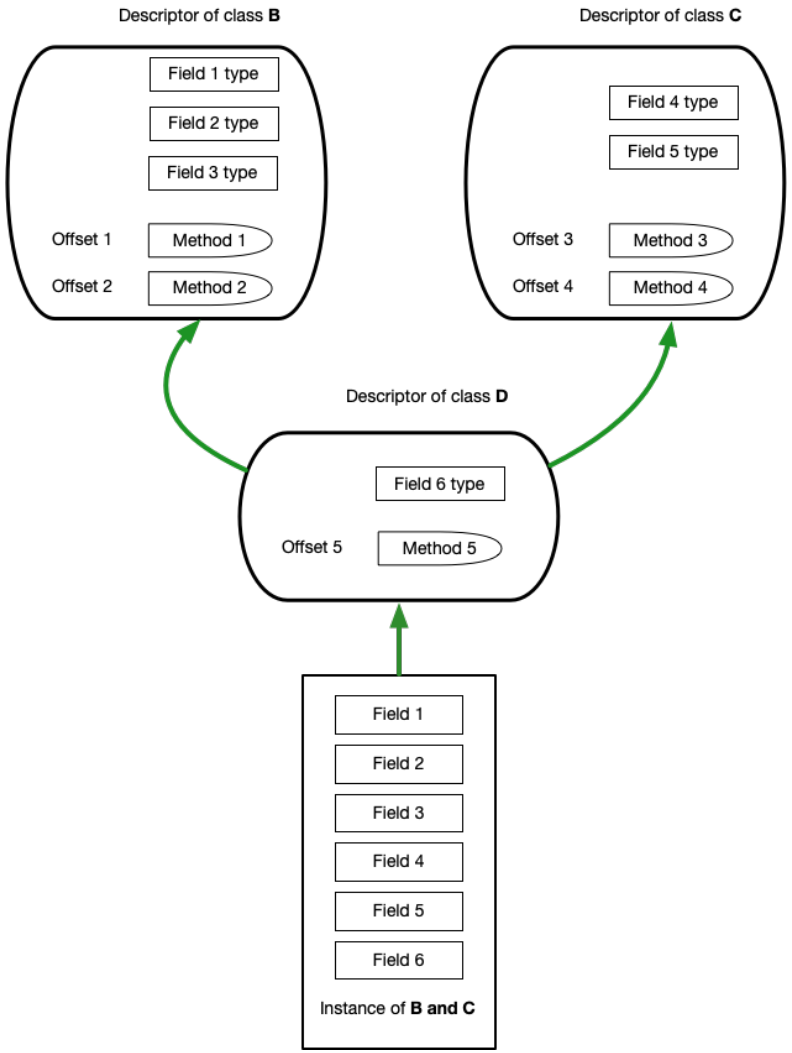
A concrete example of single inheritance

```
class A inherits Object {
  int n;
int f(int a) {return n+a;}
}
class B inherits A {
  int m;
  void g() {m *= n;}
}

A a1 = new A;
a1.n = 5;
A a2 = new A;
a2.n = 7;
B b = new B;
b.n = 9;
b.m = 13;
C c = new C;
c.n = 17;
c.m = 19;
```

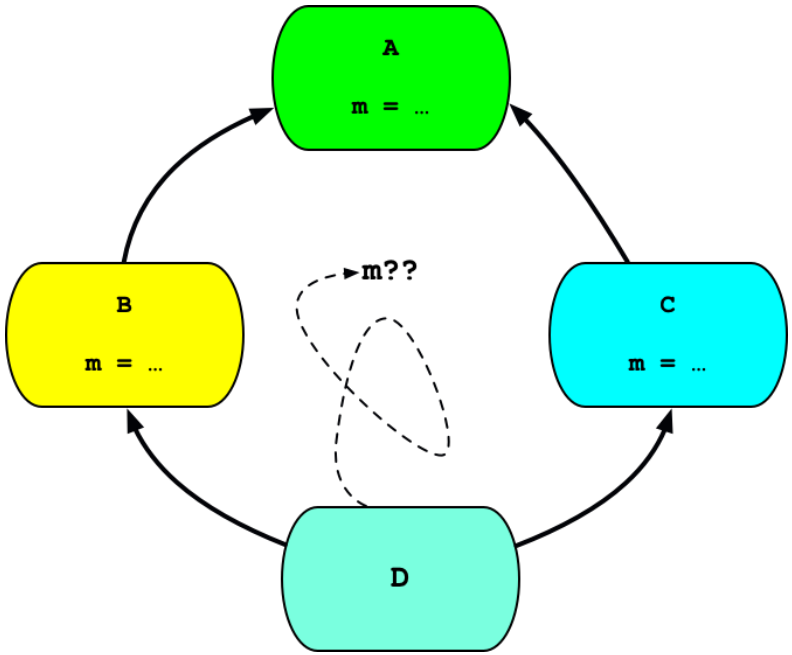


Implementing multiple inheritance



A class descriptor is shared between all objects of a class. It has pointers to the methods of the class and pointers to the descriptors of its superclasses. Since offsets are not known statically, the descriptor holds a mapping from field *names* to offsets and a mapping from method *names* to method pointers. The fields in an object do not have statically known offsets – superclass and subclass may use different offsets for the same field. The same applies for method offsets.

The diamond problem



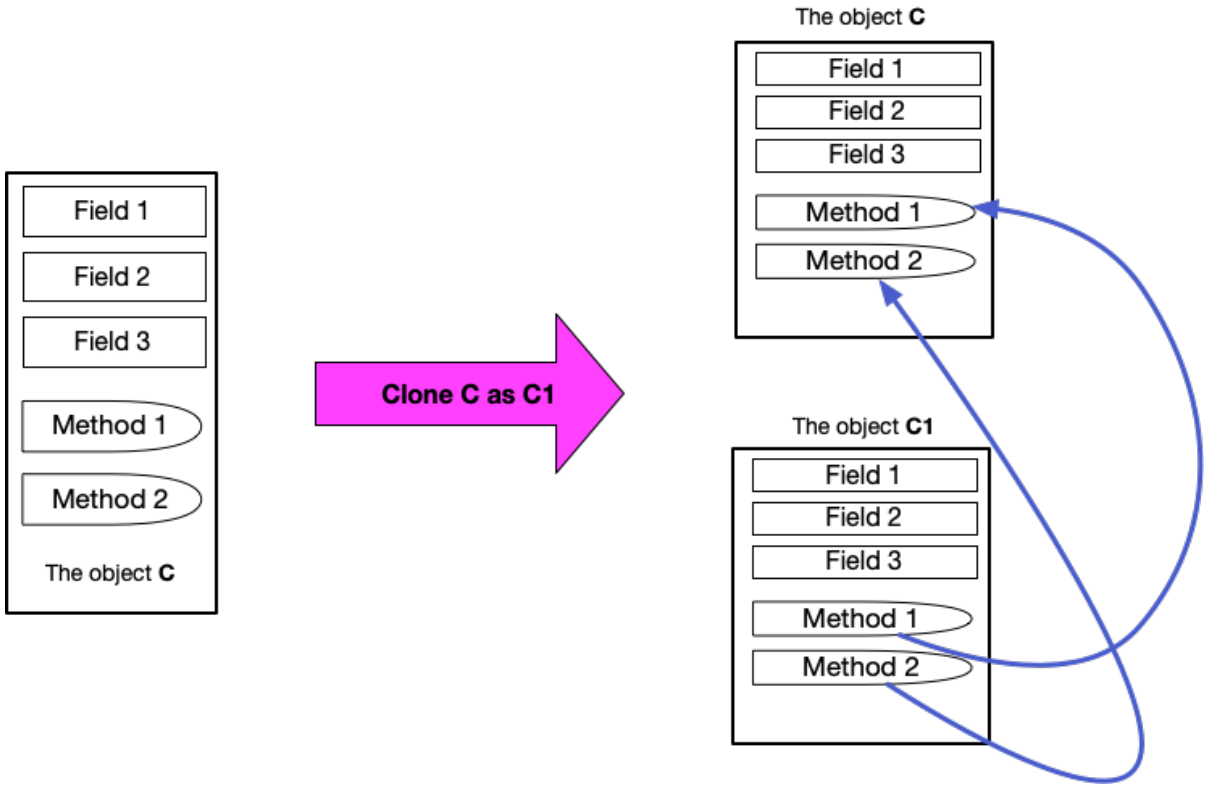
The diamond problem occurs when two classes B and C both inherit from A, and class D inherits from both B and C. If there is a method in D that is found in A but gets redefined in both B and C, which m should instances of D use?

Languages have different approaches to dealing with this.

Implementing prototype-based languages

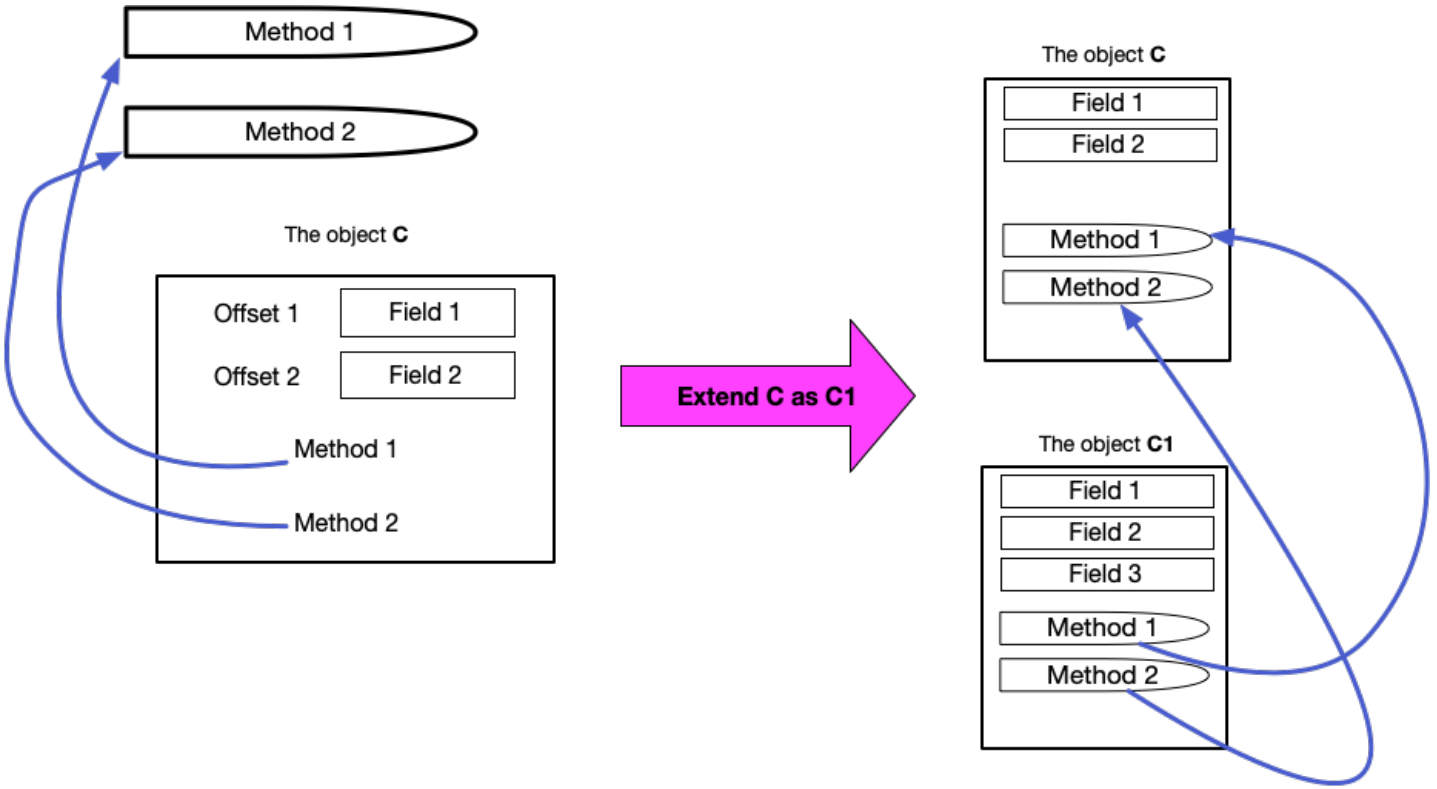
- An object is a record of fields and a pointer to a descriptor that maps field names to offsets and method names to pointers (like for multiple inheritance).
- An object can be created from scratch or by cloning an existing object (copying fields and descriptor).
- An object can be created by extending an existing object. A new descriptor is constructed at runtime.

Cloning



An object can be created from scratch or by cloning an existing object by copying its fields and descriptor.

Extension



An object can be created by extending an existing object. A new descriptor is constructed at runtime.

Part III

Prolog – A logic programming language

Logic programming languages

Logic programming began with the influential work by Alain Colmerauer in 1972. The Prolog language is due to him and his collaborators from Marseille (and Montreal). The name is an abbreviation for 'PROgrammation en LOGique'

Another influential researcher is Robert Kowalski from the University of Edinburgh and Imperial College, whose work on resolution is extremely important in this setting.

Logic programming languages

Logic programming is a declarative paradigm: A program specifies *what* conditions should hold, not *how* we make the conditions hold.

Examples of logic programming languages are Prolog, Lambda-Prolog, Mercury, Curry, CLP/R.

A program defines a family of predicates – a logical theory. The program specifies the *what*.

We run a program by performing a query that asks which values can make a proposition true in the theory. The algorithm for evaluating a query deals with the *how*.

Pure Prolog: Values

Values are terms built from variables and function symbols.

`A`	a variable (upper case)
`nil` or `42`	function symbols (lower case or number) without parameters
`cons(13,nil)`	a function symbol with two parameters that are both function symbols.
`cons(X,Y)`	a function symbol with two parameters that are both variables.

A value not containing variables is a *ground* value.

We can think of a composite term as a tree.

Pure Prolog: Propositions

A proposition is a predicate symbol with values as parameters:

`true`	a predicate symbol (lower case) without parameters
`false`	a predicate symbol without parameters
`pythagorean(3,4,5)`	a predicate symbol with three parameters that are all function symbols.
`pythagorean(A,B,C)`	a predicate symbol with three parameters that are all variables.

Distinguised from values by context.

Pure Prolog: Clauses

A clause is logical statement that is part of the theory set up by the program. It consists of a head and a body consisting of propositions. If the body is empty, we call the clause a fact. Other clauses, that are not facts, are often called rules.

The head and the body are separated by `:-` and terminated by `.`

A clause

$$p(\vec{X}) :- q_1(\vec{Y}_1), \ldots q_k(\vec{Y}_k).$$

(where the X's and Y's are sequences of variables) should be read as saying that

If $q_1(\vec{Y}_1), \ldots q_k(\vec{Y}_k)$ *all hold, then* $p(\vec{X})$ *holds as well.*

So we should think of $:-$ as implication.

Pure Prolog: Programs

A program is a list of clauses. Here is a program defining Lisp-style lists in Prolog. It has one fact and three rules.

```
list(nil).
list(cons(A,As)):- list(As).

append(nil,Bs,Bs):- list(Bs).
append(cons(A,As),Bs,cons(A,Cs)) :-
                              append(As,Bs,Cs).
```

Note that variables are local to a clause. The `Bs` in the third clause is not the same as the `Bs` in the last clause!

Pure Prolog: Programs

In our program

```
list(nil).
list(cons(A,As)):- list(As).

append(nil,Bs,Bs):- list(Bs).
append(cons(A,As),Bs,cons(A,Cs)) :-
                              append(As,Bs,Cs).
```

the first two clauses define what valid lists look like.

The last two clauses define the append predicate. The idea is that `append(L1,L2,L3)` holds if `L3` is the list that we get by appending `L1` and `L2`.

Part IV

Performing queries in Prolog

Pure Prolog: Queries

In Prolog, a query is a list of propositions to be tested and instantated by the program.

```
?- pythagorean(3,4,5).
?- list(A), append(A,A,A).
```

A ground query has no variables.

A non-ground query, here consisting of two propositions linked by a shared variable.

Appending lists with a query

What is the list that we get by appending `cons(1,nil)` and `cons(2,nil)`? We can find out by giving Prolog the query

```
append(cons(1,nil),cons(2,nil),L).
```

The free variable `L` is the value of the list. So running a query is actually a way of computing the unknowns. We solve in equations using unification.

We already saw the usefulness of unification in another setting, namely in Hindley-Milner type inference.

Other ways of using queries

But we could also ask a different kind of query:

> *What are the lists* `As` *and* `Bs` *that we should append in order to get* `cons(1,cons(2,cons(3,nil)))` *?*

Part V

The resolution algorithm

Remember: A query tries to find a substitution

Whenever we give the Prolog system a query for a given program, we ask for the values of the free variables in the query.

Our example showed, that this substitution is a form of goal-directed search.

In this search we are looking for a substitution that gives us the values of all the free variables in our query.

Resolving a query

When we give the Prolog system a query it uses unification to try to find a solution by a goal-directed search.

The system tries out all possible matching clauses from top to bottom. For each clause it tries to match every new query that this gives rise to. Each such match is a new goal.

Queries may fail and we may have to backtrack. It is also often possible to find more than one solution to a query.

This algorithm is called resolution. A solution to a query is usually called a resolvent.

Substitutions

A substitution is a set of bindings of variables to values:

$$\Theta = \{\mathtt{X} = \mathtt{a(Y,Y)}, \mathtt{Z} = \mathtt{Y}\}$$

The empty substitution is called ι.

We can apply a substitution to a value:

$$\begin{aligned} \Theta\ x &= w && \text{if } (x = w) \in \Theta \\ \Theta\ x &= x && \text{if } \Theta \text{ has no binding for } x \\ \Theta\ f(v_1, ..., v_n) &= && f(\Theta\ v_1, ..., \Theta\ v_n) \text{ for } n \geq 0 \end{aligned}$$

Operations on substitutions

- Composition: $(\Theta_1 \circ \Theta_2)\ v = \Theta_1\ (\Theta_2\ v)$
- Θ_1 is *more general* than Θ_2 if there exists Θ_3 such that $\Theta_2 = \Theta_3 \circ \Theta_1$.
- A *unifier* of two values v and w is a substitution Θ where $\Theta\ v = \Theta\ w$.
- A *most general unifier* (MGU) of v and w is a unifier Θ of v and w such that whenever Θ' is a unifier of v and w, Θ is more general than Θ'. Unique up to renaming.

The unification algorithm that we saw in the setting of Hindley-Milner type inference is also used here to find the most general unifier.

Matching clauses

Given a query

$?\text{-}\ p_1(x_1),\ p_2(x_2), \ldots,\ p_n(x_n)\,.$

we look for a clause in the program that uses the predicate symbol p_1, and rename variables in the clause to make sure that they do not overlap with variables in the query:

$p_1(y_0)\ \text{:-}\ q_1(y_1),\ q_2(y_2), \ldots,\ q_m(y_m)\,.$

We then unify $p_1(x_1)$ with $p_1(y_0)$ to get a unifier Θ (or failure). This substitution gives us a value of x_1. The new goals are

$?\text{-}\ \Theta\,(q_1(y_1),\ q_2(y_2), \ldots,\ q_m(y_m),\ p_2(x_2), \ldots,\ p_n(x_n))\,.$

If unification fails, we *backtrack*.

Backtracking

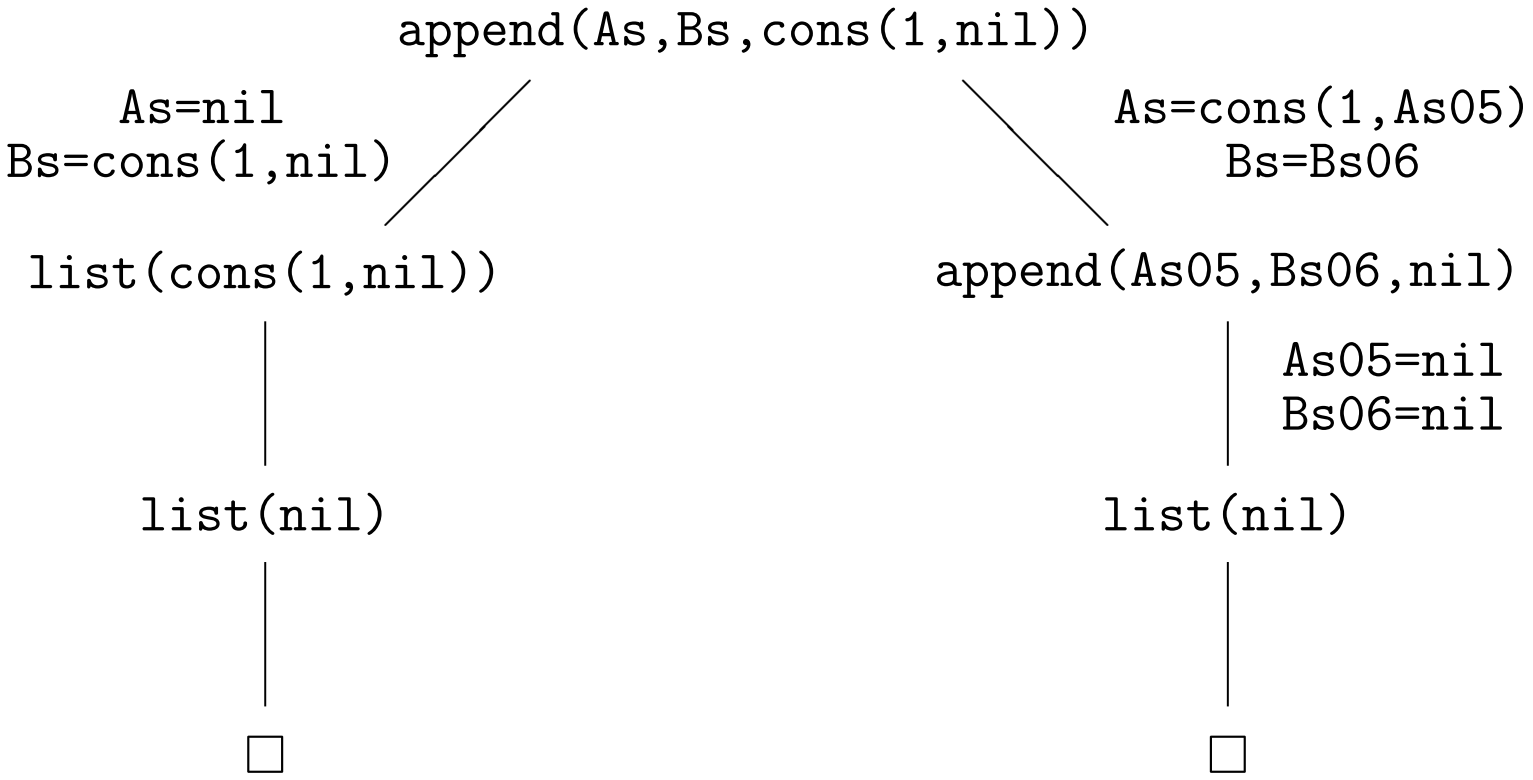
- If unification fails, we forget all bindings made by the failed unification, and try to find another matching clause.
- If there are no other clauses (we have tried all), go one step further back (undoing bindings made in between), and try the next clause there.
- Repeat until you either get successful unification or have tried all possibilities.
- If successful, continue with the remaining queries. If there are no remaining queries, return the composed substitution (restricted to variables in the original query) as your solution and ask if the user wants more solutions.
- If the user wants more solutions, backtrack.

Resolution tree

We can think of the goals that we meet during resolution as forming a tree.

- The root of the tree is the original query.
- The edges represent substitutions (restricted to variables in parent).
- Children are resolvents with *all* matching clauses for first proposition in query.
- Leafs are empty queries ($\square$), signifying solutions, or queries with no matching clauses, signifying failure.
- The tree can be infinite!

The resolution tree for `append(As,Bs,cons(1,nil))`



Finds solutions {A=nil, B=cons(1,nil)} and {A=cons(1,nil), B=nil}.

Part VI

Prolog in the real world

Implementing Pure Prolog

The resolution algorithm is complete – if a query has a resolvent, we can find it. And it is sound – we only find resolvents that actually are solutions to our query.

Soundness and completeness are often relaxed for efficiency reasons.

- The interpreter traverses the resolution tree with depth-first search (not complete).
- Omit *occurs check* (neither sound nor complete).

This means that we will not find all valid solutions, and that we may find "solutions" that are circular.

Full Prolog

Added features include:

- Cut (`!`).
  Prevents backtracking of already successful propositions.
  Reduces completeness, and variables in answers are no longer universally quantified.
- Negation as failure (`\+`).
  Not true negation: $\exists X : \backslash\!+\, p(X) \equiv \neg\,(\exists X : p(X))$.
  Implemented using cut. Sometimes postponed until all variables are bound to ground values, so quantification is not an issue.
- Arithmetic on numbers.
  Only works on fully instantiated expressions. Run-time error message if not.

Other logic programming languages

- Constraint logic languages:
  - CLP/R (constraints over real numbers)
  - Bit-Prolog (constraints over boolean variables)
- Functional-logical languages:
  - Lambda-Prolog has polymorphic typing, modular programming, and higher-order programming
  - Curry is a functional logic programming language, based on Haskell language.
  - Visual Prolog adds features from modern functional languages such as parametric polymorphism – and is object-oriented, too!

Who needs Prolog?

Prolog is extremely useful in the world of machine intelligence. Many problems in AI can easily be presented in a rule-based format.

In the 1980s, Japanese researchers worked on building a computer that was to have Prolog as its machine language. They did not quite succeed.

But Prolog is also useful when we study type systems and program semantics, as many of the definitions and properties that we study here are declarative and can naturally be expressed in a rule-based format.