Logic Programming & Type Theory

Velva Singleton

Liza Saxton

First Edition, 2012

ISBN 978-81-323-1287-1

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Logic Programming

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's [1958] advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

However, logic programming, in the narrower sense in which it is more commonly understood, is the use of logic as both a declarative and procedural representation language. It is based upon the fact that a backwards reasoning theorem-prover applied to declarative sentences in the form of implications:

If B_1 and … and B_n then H

treats the implications as goal-reduction procedures:

to show/solve H, show/solve B_1 and … and B_n.

For example, it treats the implication:

If you press the alarm signal button,
then you alert the driver of the train of a possible emergency

as the procedure:

To alert the driver of the train of a possible emergency,
press the alarm signal button.

Note that this is consistent with the BHK interpretation of constructive logic, where implication would be interpreted as a solution of problem H given solutions of B_1 … B_n. However, the defining feature of logic programming is that sets of formulas can be regarded as programs and *proof search* can be given a computational meaning. This is

achieved by restricting the underlying logic to a "well-behaved" fragment such as Horn clauses or Hereditary Harrop formulas.

As in the purely declarative case, the programmer is responsible for ensuring the truth of programs. But since automated proof search is generally infeasible, logic programming as commonly understood *also* relies on the programmer to ensure that inferences are generated efficiently. In many cases, to achieve efficiency, one needs to be aware of and to exploit the problem-solving behavior of the theorem-prover. In this respect, logic programming is comparable to conventional imperative programming; using programs to control the behaviour of a program executor. However, unlike conventional imperative programs, which have only a procedural interpretation, logic programs also have a declarative, logical interpretation, which helps to ensure their correctness. Moreover, such programs, being declarative, are at a higher conceptual level than purely imperative programs; and their program executors, being theorem-provers, operate at a higher conceptual level than conventional compilers and interpreters.

History

Logic programming in the first and wider sense gave rise to a number of implementations, such as those by Fischer Black (1964), James Slagle (1965) and Cordell Green (1969), which were question-answering systems in the spirit of McCarthy's advice-taker. Foster and Elcock's Absys (1969), on the other hand, was probably the first language to be explicitly developed as an assertional programming language.

Logic programming in the narrower sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Advocates of declarative representations were notably working at Stanford, associated with John McCarthy, Bertram Raphael and Cordell Green, and in Edinburgh, with J. Alan Robinson (an academic visitor from Syracuse University), Pat Hayes, and Bob Kowalski. Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert.

Although it was based on logic, Planner, developed at MIT, was the first language to emerge within this proceduralist paradigm [Hewitt, 1969]. Planner featured pattern directed invocation of procedural plans from goals (i.e. forward chaining) and from assertions (i.e. backward chaining). The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time. In order to cope with the very limited memory systems that were available when it was developed, Planner used backtracking control structure so that only one possible computation path had to be stored at a time. From Planner there developed the programming languages QA-4, Popler, Conniver, QLISP, and the concurrent language Ether.

Hayes and Kowalski in Edinburgh tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. Hayes (1973) developed

an equational language, Golux, in which different procedures could be obtained by altering the behavior of the theorem prover. Kowalski, on the other hand, showed how SL-resolution treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design and implementation of the programming language Prolog. From Prolog there developed, among others, the programming languages ALF, Fril, Gödel, Mercury, Oz, Ciao, Visual Prolog, XSB, and λProlog, as well as a variety of concurrent logic programming languages, constraint logic programming languages and datalog.

In 1997, the Association of Logic Programming bestowed to fifteen recognized researchers in logic programming the title *Founders of Logic Programming* to recognize them as pioneers in the field. The individuals receiving this honor were: Maurice Bruynooghe (Belgium), Jacques Cohen (USA), Alain Colmerauer (France), Keith Clark (UK), Veronica Dahl (Canada/Argentina), Maarten van Emden (Canada), Herve Gallaire (France), Robert Kowalski (UK), Jack Minker (USA), Fernando Pereira (USA), Luis Moniz Pereira (Portugal), Ray Reiter (Canada), Alan Robinson (USA), Peter Szeredi (Hungary), and David H.D. Warren (UK).

Prolog

The programming language Prolog was developed in 1972 by Alain Colmerauer. It emerged from a collaboration between Colmerauer in Marseille and Robert Kowalski in Edinburgh. Colmerauer was working on natural language understanding, using logic to represent semantics and using resolution for question-answering. During the summer of 1971, Colmerauer and Kowalski discovered that the clausal form of logic could be used to represent formal grammars and that resolution theorem provers could be used for parsing. They observed that some theorem provers, like hyper-resolution, behave as bottom-up parsers and others, like SL-resolution (1971), behave as top-down parsers.

It was in the following summer of 1972, that Kowalski, again working with Colmerauer, developed the procedural interpretation of implications. This dual declarative/procedural interpretation later became formalised in the Prolog notation

$$H :- B_1, \ldots, B_n.$$

which can be read (and used) both declaratively and procedurally. It also became clear that such clauses could be restricted to definite clauses or Horn clauses, where H, B_1, …, B_n are all atomic predicate logic formulae, and that SL-resolution could be restricted (and generalised) to LUSH or SLD-resolution. Kowalski's procedural interpretation and LUSH were described in a 1973 memo, published in 1974.

Colmerauer, with Philippe Roussel, used this dual interpretation of clauses as the basis of Prolog, which was implemented in the summer and autumn of 1972. The first Prolog program, also written in 1972 and implemented in Marseille, was a French question-answering system. The use of Prolog as a practical programming language was given great momentum by the development of a compiler by David Warren in Edinburgh in

1977. Experiments demonstrated that Edinburgh Prolog could compete with the processing speed of other symbolic programming languages such as Lisp. Edinburgh Prolog became the *de facto* standard and strongly influenced the definition of ISO standard Prolog.

Negation as failure

Micro-Planner had a construct, called "thnot", which when applied to an expression returns the value true if (and only if) the evaluation of the expression fails. An equivalent operator is normally built-in in modern Prolog's implementations and has been called "negation as failure". It is normally written as not(p), where p is an atom whose variables normally have been instantiated by the time not(p) is invoked. A more cryptic (but standard) syntax is \+ p . Negation as failure literals can occur as conditions not(B_i) in the body of program clauses.

The logical status of negation as failure was unresolved until Keith Clark [1978] showed that, under certain natural conditions, it is a correct (and sometimes complete) implementation of classical negation with respect to the completion of the program. Completion amounts roughly to regarding the set of all the program clauses with the same predicate on the left hand side, say

```
H :- Body1.
…
H :- Bodyk.
```

as a definition of the predicate

```
H iff (Body1 or … or Bodyk)
```

where "iff" means "if and only if". Writing the completion also requires explicit use of the equality predicate and the inclusion of a set of appropriate axioms for equality. However, the implementation of negation by failure needs only the if-halves of the definitions without the axioms of equality.

The notion of completion is closely related to McCarthy's circumscription semantics for default reasoning, and to the closed world assumption.

As an alternative to the completion semantics, negation as failure can also be interpreted epistemically, as in the stable model semantics of answer set programming. In this interpretation not(B_i) means literally that B_i is not known or not believed. The epistemic interpretation has the advantage that it can be combined very simply with classical negation, as in "extended logic programming", to formalise such phrases as "the contrary can not be shown", where "contrary" is classical negation and "can not be shown" is the epistemic interpretation of negation as failure.

Problem solving

In the simplified, propositional case in which a logic program and a top-level atomic goal contain no variables, backward reasoning determines an and-or tree, which constitutes the search space for solving the goal. The top-level goal is the root of the tree. Given any node in the tree and any clause whose head matches the node, there exists a set of child nodes corresponding to the sub-goals in the body of the clause. These child nodes are grouped together by an "and". The alternative sets of children corresponding to alternative ways of solving the node are grouped together by an "or".

Any search strategy can be used to search this space. Prolog uses a sequential, last-in-first-out, backtracking strategy, in which only one alternative and one sub-goal is considered at a time. Other search strategies, such as parallel search, intelligent backtracking, or best-first search to find an optimal solution, are also possible.

In the more general case, where sub-goals share variables, other strategies can be used, such as choosing the subgoal that is most highly instantiated or that is sufficiently instantiated so that only one procedure applies. Such strategies are used, for example, in concurrent logic programming.

The fact that there are alternative ways of executing a logic program has been characterised by the equation:

Algorithm = Logic + Control

where "Logic" represents a logic program and "Control" represents different theorem-proving strategies.

Knowledge representation

The fact that Horn clauses can be given a procedural interpretation and, vice versa, that goal-reduction procedures can be understood as Horn clauses + backward reasoning means that logic programs combine declarative and procedural representations of knowledge. The inclusion of negation as failure means that logic programming is a kind of non-monotonic logic.

Despite its simplicity compared with classical logic, this combination of Horn clauses and negation as failure has proved to be surprisingly expressive. For example, it has been shown to correspond, with some further extensions, quite naturally to the semi-formal language of legislation. It is also a natural language for expressing common-sense laws of cause and effect, as in the situation calculus and event calculus.

Abductive logic programming

Abductive Logic Programming is an extension of normal Logic Programming that allows some predicates, declared as abducible predicates, to be incompletely defined. Problem

solving is achieved by deriving hypotheses expressed in terms of the abducible predicates as solutions of problems to be solved. These problems can be either observations that need to be explained (as in classical abductive reasoning) or goals to be achieved (as in normal logic programming). It has been used to solve problems in Diagnosis, Planning, Natural Language and Machine Learning. It has also been used to interpret Negation as Failure as a form of abductive reasoning.

Metalogic programming

Because mathematical logic has a long tradition of distinguishing between object language and metalanguage, logic programming also allows metalevel programming. The simplest metalogic progam is the so-called "vanilla" meta-interpreter:

```
solve(true).
solve((A,B)):- solve(A),solve(B).
solve(A):- clause(A,B),solve(B).
```

where true represents an empty conjunction, and clause(A,B) means there is an object-level clause of the form A :- B.

Metalogic programming allows object-level and metalevel representations to be combined, as in natural language. It can also be used to implement any logic that is specified by means of inference rules.

Constraint logic programming

Constraint logic programming is an extension of normal Logic Programming that allows some predicates, declared as constraint predicates, to occur as literals in the body of clauses. These literals are not solved by goal-reduction using program clauses, but are added to a store of constraints, which is required to be consistent with some built-in semantics of the constraint predicates.

Problem solving is achieved by reducing the initial problem to a satisfiable set of constraints. Constraint logic programming has been used to solve problems in such fields as civil engineering, mechanical engineering, digital circuit verification, automated timetabling, air traffic control, and finance. It is closely related to abductive logic programming.

Concurrent logic programming

Keith Clark, Steve Gregory, Vijay Saraswat, Udi Shapiro, Kazunori Ueda, etc. developed a family of Prolog-like concurrent message passing systems using unification of shared variables and data structure streams for messages. Efforts were made to base these systems on mathematical logic, and they were used as the basis of the Japanese Fifth Generation Project (ICOT). However, the Prolog-like concurrent systems were based on message passing and consequently were subject to the same indeterminacy as other

concurrent message-passing systems, such as Actors. Consequently, the ICOT languages were not based on logic in the sense that computational steps could not be logically deduced [Hewitt and Agha, 1988].

Concurrent constraint logic programming combines concurrent logic programming and constraint logic programming, using constraints to control concurrency. A clause can contain a guard, which is a set of constraints that may block the applicability of the clause. When the guards of several clauses are satisfied, concurrent constraint logic programming makes a committed choice to the use of only one.

Inductive logic programming

Inductive logic programming is concerned with generalizing positive and negative examples in the context of background knowledge. Generalizations, as well as the examples and background knowledge, are expressed in logic programming syntax. Recent work in this area, combining logic programming, learning and probability, has given rise to the new field of statistical relational learning and probabilistic inductive logic programming.

Higher-order logic programming

Several researchers have extended logic programming with higher-order programming features derived from higher-order logic, such as predicate variables. Such languages include the Prolog extensions HiLog and λProlog.

Chapter 2

Prolog

Prolog	
Paradigm	Logic programming
Appeared in	1972
Designed by	Alain Colmerauer
Major implementations	Amzi! Prolog, BProlog, Ciao Prolog, ECLiPSe, GNU Prolog, Logic Programming Associates, Poplog Prolog, P#, Quintus, SICStus, Strawberry, SWI-Prolog, tuProlog, YAP-Prolog
Dialects	ISO Prolog, Edinburgh Prolog
Influenced	Visual Prolog, Mercury, Oz, Erlang, Strand, KL0, KL1, Datalog

Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in formal logic, and unlike many other programming languages, Prolog is declarative: The program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Prolog was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. While initially aimed at natural language processing, the language has since

then stretched far into other areas like theorem proving, expert systems, games, automated answering systems, ontologies and sophisticated control systems. Modern Prolog environments support creating graphical user interfaces, as well as administrative and networked applications.

Prolog syntax and semantics

The **syntax and semantics of the Prolog programming language** is the set of rules that defines how a Prolog program is written and how it is interpreted. The rules are laid out in ISO standard ISO/IEC 13211 although there are differences in the Prolog implementations.

Data types

Prolog's single data type is the *term*. Terms are either *atoms*, *numbers*, *variables* or *compound terms*.

An **atom** is a general-purpose name with no inherent meaning. It is composed of a sequence of characters that is parsed by the Prolog reader as a single unit. Atoms are usually bare words in Prolog code, written with no special syntax. However, atoms containing spaces or certain other special characters must be surrounded by single quotes. Atoms beginning with a capital letter must also be quoted, to distinguish them from variables. The empty list, written `[]`, is also an atom. Other examples of atoms include `x`, `blue`, `'Taco'`, and `'some atom'`.

Numbers can be floats or integers. Many Prolog implementations also provide unbounded integers and rational numbers.

Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms. A variable can become instantiated (bound to equal a specific term) via unification. A single underscore (_) denotes an anonymous variable and means "any term". Unlike other variables, the underscore does not represent the same value everywhere it occurs within a predicate definition.

A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero.

Examples of compound terms are `truck_year('Mazda', 1986)` and `'Person_Friends'(zelda,[tom,jim])`. Compound terms with functors that are declared as operators can be written in prefix or infix notation. For example, the terms `-(z)`, `+(a,b)` and `=(X,Y)` can also be written as `-z`, `a+b` and `X=Y`, respectively. Users can

declare arbitrary functors as operators with different precedences to allow for domain-specific notations. The notation *f/n* is commonly used to denote a term with functor *f* and arity *n*.

Special cases of compound terms:

- *Lists* are defined inductively: The atom `[]` is a list. A compound term with functor `.` (dot) and arity 2, whose second argument is a list, is itself a list. There exists special syntax for denoting lists: `.(A, B)` is equivalent to `[A|B]`. For example, the list `.(1, .(2, .(3, [])))` can also be written as `[1 | [2 | [3 | []]]]`, or even more compactly as `[1,2,3]`.
- *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding or Unicode if the system supports Unicode.

Prolog programs

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses, a Turing-complete subset of first-order predicate logic. There are two types of clauses: Facts and rules. A rule is of the form

```
Head :- Body.
```

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's **goals**. The built-in predicate `,/2` (meaning a 2-arity operator with name `,`) denotes conjunction of goals, and `;/2` denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called **facts**. An example of a fact is:

```
cat(tom).
```

which is equivalent to the rule:

```
cat(tom) :- true.
```

The built-in predicate `true/0` is always true.

Evaluation

Execution of a Prolog program is initiated by the user's posting of a single goal, called the query. Logically, the Prolog engine tries to find a resolution refutation of the negated query. The resolution method used by Prolog is called SLD resolution. If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, all generated variable bindings are reported to the user, and the query is said to have succeeded. Operationally,

Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking. For example:

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

This is obtained as follows: Initially, the only matching clause-head for the query `sibling(sally, erica)` is the first one, so proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction `(parent_child(Z,sally), parent_child(Z,erica))`. The next goal to be proved is the leftmost one of this conjunction, i.e., `parent_child(Z, sally)`. Two clause heads match this goal. The system creates a choice-point and tries the first alternative, whose body is `father_child(Z, sally)`. This goal can be proved using the fact `father_child(tom, sally)`, so the binding `Z = tom` is generated, and the next goal to be proved is the second part of the above conjunction: `parent_child(tom, erica)`. Again, this can be proved by the corresponding fact. Since all goals could be proved, the query succeeds. Since the query contained no variables, no bindings are reported to the user. A query with variables, like:

```
?- father_child(Father, Child).
```

enumerates all valid answers on backtracking.

Notice that with the code as stated above, the query `?- sibling(sally, sally).` also succeeds. One would insert additional goals to describe the relevant restrictions, if desired.

Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates. Prolog systems typically implement a well-known optimization technique called tail call optimization (TCO) for deterministic predicates exhibiting tail recursion or, more generally, tail calls: A clause's stack frame is discarded before performing a call in a tail position. Therefore, deterministic tail-recursive predicates are executed with constant stack space, like loops in other languages.

Negation

The built-in Prolog predicate `\+/1` provides negation as failure, which allows for non-monotonic reasoning. The goal `\+ illegal(X)` in the rule

```
legal(X) :- \+ illegal(X).
```

is evaluated as follows: Prolog attempts to prove the `illegal(X)`. If a proof for that goal can be found, the original goal (i.e., `\+ illegal(X)`) fails. If no proof can be found, the original goal succeeds. Therefore, the `\+/1` prefix operator is called the "not provable" operator, since the query `?- \+ Goal.` succeeds if Goal is not provable. This kind of negation is sound if its argument is "ground" (i.e. contains no variables). Soundness is lost if the argument contains variables. In particular, the query `?- legal(X).` can now not be used to enumerate all things that are legal.

Semantics

Under a declarative reading, the order of rules, and of goals within rules, is irrelevant since logical disjunction and conjunction are commutative. Procedurally, however, it is often important to take into account Prolog's execution strategy, either for efficiency reasons, or due to the semantics of impure built-in predicates for which the order of evaluation matters. Also, as Prolog interpreters try to unify clauses in the order they're provided, failing to give a correct ordering can lead to infinite recursion, as in:

```
predicate1(X) :-
  predicate2(X,X).
predicate2(X,Y) :-
  predicate1(X),
  X \= Y.
```

Given this ordering, any query of the form

```
?- predicate1(atom).
```

will recur till the stack is exhausted. If, however, the last 3 lines were changed to:

```
predicate2(X,Y) :-
  X \= Y,
```

```
predicate1(X).
```

the same query would lead to a No. outcome in a very short time.

Definite clause grammars

There is a special notation called definite clause grammars (DCGs). A rule defined via `-->/2` instead of `:-/2` is expanded by the preprocessor (`expand_term/2`, a facility analogous to macros in other languages) according to a few straightforward rewriting rules, resulting in ordinary Prolog clauses. Most notably, the rewriting equips the predicate with two additional arguments, which can be used to implicitly thread state around, analogous to monads in other languages. DCGs are often used to write parsers or list generators, as they also provide a convenient interface to list differences.

Parser example

A larger example will show the potential of using Prolog in parsing.

Given the sentence expressed in Backus-Naur Form:

```
<sentence>    ::=  <stat_part>
<stat_part>   ::=  <statement> | <stat_part> <statement>
<statement>   ::=  <id> = <expression> ;
<expression>  ::=  <operand> | <expression> <operator> <operand>
<operand>     ::=  <id> | <digit>
<id>          ::=  a | b
<digit>       ::=  0..9
<operator>    ::=  + | - | *
```

This can be written in Prolog using DCGs, corresponding to a predictive parser with one token look-ahead:

```
sentence(S)                --> statement(S0), sentence_r(S0, S).
sentence_r(S, S)           --> [].
sentence_r(S0, seq(S0, S)) --> statement(S1), sentence_r(S1, S).

statement(assign(Id,E)) --> id(Id), [=], expression(E), [;].

expression(E) --> term(T), expression_r(T, E).
expression_r(E, E)  --> [].
expression_r(E0, E) --> [+], term(T), expression_r(plus(E0,T), E).
expression_r(E0, E) --> [-], term(T), expression_r(minus(E0, T), E).

term(T)       --> factor(F), term_r(F, T).
term_r(T, T)  --> [].
term_r(T0, T) --> [*], factor(F), term_r(times(T0, F), T).

factor(id(ID))   --> id(ID).
factor(digit(D)) --> [D], { (number(D) ; var(D)), between(0, 9, D)}.

id(a) --> [a].
```

```
id(b) --> [b].
```

This code defines a relation between a sentence (given as a list of tokens) and its abstract syntax tree (AST). Example query:

```
?- phrase(sentence(AST), [a,=,1,+,3,*,b,;,b,=,0,;]).
AST = seq(assign(a, plus(digit(1), times(digit(3), id(b)))), assign(b,
digit(0))) ;
```

The AST is represented using Prolog terms and can be used to apply optimizations, to compile such expressions to machine-code, or to directly interpret such statements. As is typical for the relational nature of predicates, these definitions can be used both to parse and generate sentences, and also to check whether a given tree corresponds to a given list of tokens. Using iterative deepening for fair enumeration, each arbitrary but fixed sentence and its corresponding AST will be generated eventually:

```
?- length(Tokens, _), phrase(sentence(AST), Tokens).
 Tokens = [a, =, a, (;)], AST = assign(a, id(a)) ;
 Tokens = [a, =, b, (;)], AST = assign(a, id(b))
 etc.
```

Data types

Prolog's single data type is the *term*. Terms are either *atoms*, *numbers*, *variables* or *compound terms*.

- An **atom** is a general-purpose name with no inherent meaning. Examples of atoms include `x`, `blue`, `'Taco'`, and `'some atom'`.
- **Numbers** can be floats or integers.
- **Variables** are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. Examples of compound terms are `truck_year('Mazda', 1986)` and `'Person_Friends'(zelda,[tom,jim])`.

Special cases of compound terms:

- A *List* is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, `[]`. For example `[1,2,3]` or `[red,green,blue]`.
- *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding or Unicode if the system supports Unicode. For example, `"to be, or not to be"`.

Rules and facts

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: Facts and rules. A rule is of the form

```
Head :- Body.
```

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's **goals**. The built-in predicate `,/2` (meaning a 2-arity operator with name `,`) denotes conjunction of goals, and `;/2` denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called **facts**. An example of a fact is:

```
cat(tom).
```

which is equivalent to the rule:

```
cat(tom) :- true.
```

The built-in predicate `true/0` is always true.

Given the above fact, one can ask:

is tom a cat?

```
?- cat(tom).
Yes
```

what things are cats?

```
?- cat(X).
X = tom
```

Due to the relational nature of many built-in predicates, they can typically be used in several directions. For example, `length/2` can be used to determine the length of a list (`length(List, L)`, given a list `List`) as well as to generate a list skeleton of a given length (`length(X, 5)`), and also to generate both list skeletons and their lengths together (`length(X, L)`). Similarly, `append/3` can be used both to append two lists (`append(ListA, ListB, X)` given lists `ListA` and `ListB`) as well as to split a given list into parts (`append(X, Y, List)`, given a list `List`). For this reason, a comparatively small set of library predicates suffices for many Prolog programs.

As a general purpose language, Prolog also provides various built-in predicates to perform routine activities like input/output, using graphics and otherwise communicating with the operating system. These predicates are not given a relational meaning and are

only useful for the side-effects they exhibit on the system. For example, the predicate `write/1` displays a term on the screen.

Evaluation

Execution of a Prolog program is initiated by the user's posting of a single goal, called the query. Logically, the Prolog engine tries to find a resolution refutation of the negated query. The resolution method used by Prolog is called SLD resolution. If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, all generated variable bindings are reported to the user, and the query is said to have succeeded. Operationally, Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking. For example:

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

This is obtained as follows: Initially, the only matching clause-head for the query `sibling(sally, erica)` is the first one, so proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction `(parent_child(Z,sally), parent_child(Z,erica))`. The next goal to be proved is the leftmost one of this conjunction, i.e., `parent_child(Z, sally)`. Two clause heads match this goal. The system creates a choice-point and tries the first alternative, whose body is `father_child(Z, sally)`. This goal can be proved using the fact `father_child(tom, sally)`, so the binding `Z = tom` is generated, and the next goal to be proved is the second part of the above conjunction: `parent_child(tom, erica)`. Again, this can be proved by the corresponding fact. Since all goals could be proved, the query succeeds. Since the query contained no variables, no bindings are reported to the user. A query with variables, like:

```
?- father_child(Father, Child).
```

enumerates all valid answers on backtracking.

Notice that with the code as stated above, the query `?- sibling(sally, sally).` also succeeds. One would insert additional goals to describe the relevant restrictions, if desired.

Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates.

Negation

The built-in Prolog predicate `\+/1` provides negation as failure, which allows for non-monotonic reasoning. The goal `\+ illegal(X)` in the rule

```
legal(X) :- \+ illegal(X).
```

is evaluated as follows: Prolog attempts to prove the `illegal(X)`. If a proof for that goal can be found, the original goal (i.e., `\+ illegal(X)`) fails. If no proof can be found, the original goal succeeds. Therefore, the `\+/1` prefix operator is called the "not provable" operator, since the query `?- \+ Goal.` succeeds if Goal is not provable. This kind of negation is sound if its argument is "ground" (i.e. contains no variables). Soundness is lost if the argument contains variables and the proof procedure is complete. In particular, the query `?- legal(X).` can now not be used to enumerate all things that are legal.

Examples

Here follow some example programs written in Prolog.

Hello world

An example of a query:

```
?- write('Hello world!'), nl.
Hello world!
true.

?-
```

Compiler optimization

Any computation can be expressed declaratively as a sequence of state transitions. As an example, an optimizing compiler with three optimization passes could be implemented as a relation between an initial program and its optimized form:

```
program_optimized(Prog0, Prog) :-
    optimization_pass_1(Prog0, Prog1),
    optimization_pass_2(Prog1, Prog2),
    optimization_pass_3(Prog2, Prog).
```

or equivalently using DCG notation:

```
program_optimized --> optimization_pass_1, optimization_pass_2,
optimization_pass_3.
```

Quicksort

The Quicksort sorting algorithm, relating a list to its sorted version:

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])     --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Dynamic programming

The following Prolog program uses dynamic programming to find the longest common subsequence of two lists in polynomial time. The clause database is used for memoization:

```
:- dynamic(stored/1).

memo(Goal) :- ( stored(Goal) -> true ; Goal, assertz(stored(Goal)) ).

lcs([], _, []) :- !.
lcs(_, [], []) :- !.
lcs([X|Xs], [X|Ys], [X|Ls]) :- !, memo(lcs(Xs, Ys, Ls)).
lcs([X|Xs], [Y|Ys], Ls) :-
    memo(lcs([X|Xs], Ys, Ls1)), memo(lcs(Xs, [Y|Ys], Ls2)),
    length(Ls1, L1), length(Ls2, L2),
    (   L1 >= L2 -> Ls = Ls1 ; Ls = Ls2 ).
```

Example query:

```
?- lcs([x,m,j,y,a,u,z], [m,z,j,a,w,x,u], Ls).
Ls = [m, j, a, u]
```

Design patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design. In Prolog, design patterns go under various names: skeletons and techniques, cliches, program schemata, and logic description schemata. An alternative to design patterns is higher order programming.

Higher-order programming

By definition, first-order logic does not allow quantification over predicates. A higher-order predicate is a predicate that takes one or more other predicates as arguments. Prolog already has some built-in higher-order predicates such as `call/1`, `findall/3`, `setof/3`, and `bagof/3`. Furthermore, since arbitrary Prolog goals can be constructed and evaluated at run-time, it is easy to write higher-order predicates like `maplist/2`, which applies an arbitrary predicate to each member of a given list, and `sublist/3`, which filters elements that satisfy a given predicate, also allowing for currying.

To convert solutions from temporal representation (answer substitutions on backtracking) to spatial representation (terms), Prolog has various all-solutions predicates that collect all answer substitutions of a given query in a list. This can be used for list comprehension. For example, perfect numbers equal the sum of their proper divisors:

```
perfect(N) :-
    between(1, inf, N), U is N // 2,
    findall(D, (between(1,U,D), N mod D =:= 0), Ds),
    sumlist(Ds, N).
```

This can be used to enumerate perfect numbers, and also to check whether a number is perfect.

Modules

For programming in the large, Prolog provides a module system. The module system is standardised by ISO. However, not all Prolog compilers support modules and there are compatibility problems between the module systems of the major Prolog compilers. Consequently, modules written on one Prolog compiler will not necessarily work on others.

Parsing

There is a special notation called definite clause grammars (DCGs). A rule defined via `-->/2` instead of `:-/2` is expanded by the preprocessor (`expand_term/2`, a facility analogous to macros in other languages) according to a few straightforward rewriting rules, resulting in ordinary Prolog clauses. Most notably, the rewriting equips the predicate with two additional arguments, which can be used to implicitly thread state

around, analogous to monads in other languages. DCGs are often used to write parsers or list generators, as they also provide a convenient interface to list differences.

Meta-interpreters and reflection

Prolog is a homoiconic language and provides many facilities for reflection. Its implicit execution strategy makes it possible to write a concise meta-circular evaluator (also called *meta-interpreter*) for pure Prolog code. Since Prolog programs are themselves sequences of Prolog terms (`:-/2` is an infix operator) that are easily read and inspected using built-in mechanisms (like `read/1`), it is easy to write customized interpreters that augment Prolog with domain-specific features.

Turing completeness

Pure Prolog is based on a subset of first-order predicate logic, Horn clauses, which is Turing-complete. Turing completeness of Prolog can be shown by using it to simulate a Turing machine:

```
turing(Tape0, Tape) :-
    perform(q0, [], Ls, Tape0, Rs),
    reverse(Ls, Ls1),
    append(Ls1, Rs, Tape).

perform(qf, Ls, Ls, Rs, Rs) :- !.
perform(Q0, Ls0, Ls, Rs0, Rs) :-
    symbol(Rs0, Sym, RsRest),
    once(rule(Q0, Sym, Q1, NewSym, Action)),
    action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1),
    perform(Q1, Ls1, Ls, Rs1, Rs).

symbol([], b, []).
symbol([Sym|Rs], Sym, Rs).

action(left, Ls0, Ls, Rs0, Rs) :- left(Ls0, Ls, Rs0, Rs).
action(stay, Ls, Ls, Rs, Rs).
action(right, Ls0, [Sym|Ls0], [Sym|Rs], Rs).

left([], [], Rs0, [b|Rs0]).
left([L|Ls], Ls, Rs, [L|Rs]).
```

A simple example Turing machine is specified by the facts:

```
rule(q0, 1, q0, 1, right).
rule(q0, b, qf, 1, stay).
```

This machine performs incrementation by one of a number in unary encoding: It loops over any number of "1" cells and appends an additional "1" at the end. Example query and result:

```
?- turing([1,1,1], Ts).
```

```
Ts = [1, 1, 1, 1] ;
```

This illustrates how any computation can be expressed declaratively as a sequence of state transitions, implemented in Prolog as a relation between successive states of interest.

Implementation

ISO Prolog

The ISO Prolog standard consists of two parts. ISO/IEC 13211-1, published in 1995, aims to standardize the existing practices of the many implementations of the core elements of Prolog. It has clarified aspects of the language that were previously ambiguous and leads to portable programs. ISO/IEC 13211-2, published in 2000, adds support for modules to the standard. The standard is maintained by the ISO/IEC JTC1/SC22/WG17 working group. ANSI X3J17 is the US Technical Advisory Group for the standard.

Compilation

For efficiency, Prolog code is typically compiled to abstract machine code, often influenced by the register-based Warren Abstract Machine (WAM) instruction set. Some implementations employ abstract interpretation to derive type and mode information of predicates at compile time, or compile to real machine code for high performance. Devising efficient implementation methods for Prolog code is a field of active research in the logic programming community, and various other execution methods are employed in some implementations. These include clause binarization and stack-based virtual machines.

Tail recursion

Prolog systems typically implement a well-known optimization method called tail call optimization (TCO) for deterministic predicates exhibiting tail recursion or, more generally, tail calls: A clause's stack frame is discarded before performing a call in a tail position. Therefore, deterministic tail-recursive predicates are executed with constant stack space, like loops in other languages.

Tabling

Some Prolog systems, (BProlog, XSB and Yap), implement a memoization method called *tabling*, which frees the user from manually storing intermediate results.

Implementation in hardware

During the Fifth Generation Computer Systems project, there were attempts to implement Prolog in hardware with the aim of achieving faster execution with dedicated

architectures. Furthermore, Prolog has a number of properties that may allow speed-up through parallel execution. A more recent approach has been to compile restricted Prolog programs to a field programmable gate array. However, rapid progress in general-purpose hardware has consistently overtaken more specialised architectures.

Criticism

Although Prolog is widely used in research and education, Prolog and other logic programming languages have not had a significant impact on the computer industry in general. Most applications are small by industrial standards with few exceeding 100,000 lines of code. Programming in the large is considered to be complicated because not all Prolog compilers support modules, and there are compatibility problems between the module systems of the major Prolog compilers. Portability of Prolog code across implementations has also been a problem but developments since 2007 have meant: "the portability within the family of Edinburgh/Quintus derived Prolog implementations is good enough to allow for maintaining portable real-world applications."

Software developed in Prolog has been criticised for having a high performance penalty compared to conventional programming languages. However, advances in implementation methods have reduced the penalties to as little as 25%-50% for some applications.

Extensions

Various implementations have been developed from Prolog to extend logic programming capabilities in numerous directions. These include types, modes, constraint logic programming (CLP), object-oriented logic programming (OOLP), concurrency, linear logic (LLP), functional and higher-order logic programming capabilities, plus interoperability with knowledge bases:

Types

Prolog is an untyped language. Attempts to introduce types date back to the 1980s, and as of 2008 there are still attempts to extend Prolog with types. Type information is useful not only for type safety but also for reasoning about Prolog programs.

Modes

The syntax of Prolog does not specify which arguments of a predicate are inputs and which are outputs. However, this information is significant and it is recommended that it be included in the comments. Modes provide valuable information when reasoning about Prolog programs and can also be used to accelerate execution.

Constraints

Constraint logic programming extends Prolog to include concepts from constraint satisfaction. A constraint logic program allows constraints in the body of clauses, such as: `A(X,Y) :- X+Y>0.` It is suited to large-scale combinatorial optimisation problems. and is thus useful for applications in industrial settings, such as automated time-tabling and production scheduling. Most Prolog systems ship with at least one constraint solver for finite domains, and often also with solvers for other domains like rational numbers.

Higher-order programming

HiLog and λProlog extend Prolog with higher-order programming features.

Logtalk

Logtalk

Paradigm	Logic programming
Appeared in	1998
Designed by	Paulo Moura
Stable release	2.42.0 (December 1, 2010; 9 days ago)
Influenced by	Prolog, Smalltalk; logic programming, object-oriented programming, prototype-based programming
OS	Cross-platform
License	Artistic License 2.0

Logtalk is an object-oriented logic programming language that extends the Prolog language with a feature set suitable for programming in the large. It provides support for encapsulation and data hiding, separation of concerns and enhanced code reuse. Logtalk uses standard Prolog syntax with the addition of a few operators and directives.

It is distributed under an open source license and can run using ISO-compliant Prolog implementations as the back-end compiler.

Features

Logtalk aims to bring together the advantages of object-oriented programming and logic programming. Object-orientation emphasizes developing discrete, reusable units of software, while logic programming emphasizes representing the knowledge of each object in a declarative way.

As an object-oriented programming language, Logtalk's major features include support for both classes (with optional metaclasses) and prototypes, parametric objects, protocols (interfaces), categories (mixins, aspects), multiple inheritance, event-driven programming, high-level multi-threading programming, reflection, and automatic generation of documentation.

For Prolog programmers, Logtalk provides predicate namespaces (supporting both static and dynamic objects), private, protected, and public object predicates, coinductive predicates, separation between interface and implementation, better portability than Prolog modules, simple and intuitive meta-predicate semantics, and lambda expressions.

Examples

Logtalk's syntax is based on Prolog:

```
?- write('Hello world'), nl.
Hello world
true.
```

Defining an object:

```
:- object(my_first_object).

    :- public(p1/0).
    p1 :- write('This is a public predicate'), nl.

    :- private(p2/0).
    p2 :- write('This is a private predicate'), nl.

:- end_object.
```

Using the object:

```
?- my_first_object::p1.
This is a public predicate
true.
```

Trying to access the private predicate gives an error:

```
?- my_first_object::p2.
ERROR: error(permission_error(access, private_predicate, p2),
my_first_object::p2, user)
```

Prolog back-end compatibility

As of December 2010, supported back-end Prolog compilers include B-Prolog, CxProlog, ECLiPSe, GNU Prolog, Qu-Prolog, SICStus Prolog, SWI-Prolog, XSB, and YAP Prolog. Logtalk allows seamless use of the back-end Prolog compiler libraries from within object and categories.

Developer tools

Logtalk features on-line help, an entity diagram generator tool, a built-in debugger (based on an extended version of the traditional Procedure Box model found on most Prolog compilers), and is also compatible with selected back-end Prolog profilers.

Applications

Logtalk has been used to process STEP data models used to exchange product manufacturing information. It has also been used to implement a reasoning system that allows preference reasoning and constraint solving.

Oblog is a small, portable, object-oriented extension to Prolog by Margaret McDougall of EdCAAD, University of Edinburgh.

Graphics

Prolog systems that provide a graphics library are SWI-prolog, Visual-prolog and B-Prolog.

Concurrency

Prolog-MPI is an open-source SWI-Prolog extension for distributed computing over the Message Passing Interface. Also there are various concurrent Prolog programming languages.

Web programming

Some Prolog implementations, notably SWI-Prolog, support server-side web programming with support for web protocols, HTML and XML. There are also extensions to support semantic web formats such as RDF and OWL. Prolog has also been suggested as a client-side language.

Adobe Flash

Cedar is a free and basic Prolog interpreter. From version 4 and above Cedar has a FCA (Flash Cedar App) support. This provides a new platform to programing in Prolog through ActionScript.

Other

- F-logic extends Prolog with frames/objects for knowledge representation.
- OW Prolog has been created in order to answer Prolog's lack of graphics and interface.

Interfaces to other languages

The Logic Server API allows both the extension and embedding of Prolog in C, C++, Java, VB, Delphi, .NET and any language/environment which can call a .dll or .so. It is implemented for Amzi! Prolog Amzi! Prolog + Logic Server but the API specification can be made available for any implementation. Frameworks exist which can bridge between Prolog and Java:

- JPL is a bi-directional Java Prolog bridge which ships with SWI-Prolog by default, allowing Java and Prolog to call each other respectively. It is known to have good concurrency support and is under active development.
- InterProlog, a programming library bridge between Java and Prolog, implementing bi-directional predicate/method calling between both languages. Java objects can be mapped into Prolog terms and vice-versa. Allows the development of GUIs and other functionality in Java while leaving logic processing in the Prolog layer. Supports XSB, SWI-Prolog and YAP.
- Prova provides native syntax integration with Java, agent messaging and reaction rules. Prova positions itself as a rule-based scripting (RBS) system for middleware. The language breaks new ground in combining imperative and declarative programming.
- PROL An embeddable Prolog engine for Java. It includes a small IDE and a few libraries.
- GNU Prolog for Java is an implementation of ISO Prolog as a Java library (gnu.prolog)

History

The name *Prolog* was chosen by Philippe Roussel as an abbreviation for *programmation en logique* (French for *programming in logic*). It was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. It was motivated in part by the desire to reconcile the use of logic as a declarative knowledge representation language with the procedural representation of knowledge that was popular in North America in the late 1960s and early 1970s. According to Robert Kowalski, the first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel. The first implementations of Prolog were interpreters, however, David H. D. Warren created the Warren Abstract Machine, an early and influential Prolog compiler which came to define the "Edinburgh Prolog" dialect which served as the basis for the syntax of most modern implementations.

Much of the modern development of Prolog came from the impetus of the Fifth Generation Computer Systems project (FGCS), which developed a variant of Prolog named *Kernel Language* for its first operating system.

Pure Prolog was originally restricted to the use of a resolution theorem prover with Horn clauses of the form:

H :- B_1, ..., B_n.

The application of the theorem-prover treats such clauses as procedures:

to show/solve H, show/solve B_1 and ... and B_n.

Pure Prolog was soon extended, however, to include negation as failure, in which negative conditions of the form not(B_i) are shown by trying and failing to solve the corresponding positive conditions B_i.

Subsequent extensions of Prolog by the original team introduced constraint logic programming abilities into the implementations.

Chapter 3

Negation as Failure and Stable Model Semantics

Negation as failure

Negation as failure (**NAF**, for short) is a non-monotonic inference rule in logic programming, used to derive $\mathrm{not}\ p$ (i.e. that p is assumed not to hold) from failure to derive p. Note that $\mathrm{not}\ p$ can be different from the statement $\neg p$ of the logical negation of p, depending on the completeness of the inference algorithm and thus also on the formal logic system.

Negation as failure has been an important feature of logic programming since the earliest days of both Planner and Prolog. In Prolog, it is usually implemented using Prolog's extralogical constructs.

Planner semantics

In Planner, negation as failure could be implemented as follows:

```
if (not (goal p)), then (assert ¬p)
```

which says that if the goal to prove `p` fails, then assert `¬p`. Note that the above example uses true mathematical negation, which cannot be expressed in Prolog.

Prolog semantics

In pure Prolog, NAF literals of the form $not\ p$ can occur in the body of clauses and can be used to derive other NAF literals. For example, given only the four clauses

$$p \leftarrow q \wedge \mathrm{not}\ r$$
$$q \leftarrow s$$
$$q \leftarrow t$$
$$t \leftarrow$$

NAF derives $\text{not } s$, $\text{not } r$ and p.

Completion semantics

The semantics of NAF remained an open issue until Keith Clark [1978] showed that it is correct with respect to the completion of the logic program, where, loosely speaking, "only" and ←are interpreted as "if and only if", written as "iff" or "≡".

For example, the completion of the four clauses above is

$$p \equiv q \wedge not\ r$$
$$q \equiv s \vee t$$
$$t \equiv \text{true}$$
$$r \equiv \text{false}$$
$$s \equiv \text{false}$$

The NAF inference rule simulates reasoning explicitly with the completion, where both sides of the equivalence are negated and negation on the right-hand side is distributed down to atomic formulae. For example, to show $\text{not } p$, NAF simulates reasoning with the equivalences

$$\text{not } p \equiv \text{not } q \vee r$$
$$\text{not } q \equiv \text{not } s \wedge \text{not } t$$
$$\text{not } t \equiv \text{false}$$
$$\text{not } r \equiv \text{true}$$
$$\text{not } s \equiv \text{true}$$

In the non-propositional case, the completion needs to be augmented with equality axioms, to formalise the assumption that individuals with distinct names are distinct. NAF simulates this by failure of unification. For example, given only the two clauses

$$p(a) \leftarrow$$
$$p(b) \leftarrow_{\text{t}}$$

NAF derives $\text{not } p(c)$.

The completion of the program is

$$p(X) \equiv X = a \vee X = b$$

augmented with unique names axioms and domain closure axioms.

The completion semantics is closely related both to circumscription and to the closed world assumption.

Autoepistemic semantics

The completion semantics justifies interpreting the result $\text{not } p$ of a NAF inference as the classical negation $\neg p$ of p. However, Michael Gelfond [1987] showed that it is also possible to interpret $\text{not } p$ literally as "p can not be shown", "p is not known" or "p is not believed", as in autoepistemic logic. The autoepistemic interpretation was developed further by Gelfond and Lifschitz [1988] and is the basis of answer set programming.

The autoepistemics semantics of a pure Prolog program P with NAF literals is obtained by "expanding" P with a set of ground (variable-free) NAF literals Δ that is stable in the sense that

$$\Delta = \{\text{not } p \mid p \text{ is not implied by P } \cup \Delta\}$$

In other words, a set of assumptions Δ about what can not be shown is stable if and only if Δ is the set of all sentences that truly can not be shown from the program P expanded by Δ. Here, because of the simple syntax of pure Prolog programs, "implied by" can be understood very simply as derivability using modus ponens and universal instantiation alone.

A program can have zero, one or more stable expansions. For example

$$p \leftarrow \text{not } p$$

has no stable expansions.

$$p \leftarrow \text{not } q$$

has exactly one stable expansion $\Delta = \{\text{not } q\}$

$$p \leftarrow \text{not } q$$
$$q \leftarrow \text{not } p$$

has exactly two stable expansions $\Delta_1 = \{\text{not } p\}$ and $\Delta_2 = \{\text{not } q\}$.

The autoepistemic interpretation of NAF can be combined with classical negation, as in extended logic programming and answer set programming. Combining the two negations, it is possible to express, for example

$\neg p \leftarrow \text{not } p$ (the closed world assumption) and
$p \leftarrow \text{not } \neg p$ (p holds by default).

Stable model semantics

The concept of a **stable model**, or answer set, is used to define a declarative semantics for logic programs with negation as failure. This is one of several standard approaches to the meaning of negation in logic programming, along with program completion and the well-founded semantics. The stable model semantics is the basis of answer set programming.

Motivation

Research on the declarative semantics of negation in logic programming was motivated by the fact that the behavior of SLDNF resolution -- the generalization of SLD resolution used by Prolog in the presence of negation in the bodies of rules -- does not fully match the truth tables familiar from classical propositional logic. Consider, for instance, the program

$$p$$
$$r \leftarrow p,\ q$$
$$s \leftarrow p,\ \text{not } q.$$

Given this program, the query *p* will succeed, because the program includes *p* as a fact; the query *q* will fail, because it does not occur in the head of any of the rules. The query *r* will fail also, because the only rule with *r* in the head contains the subgoal *q* in its body; as we have seen, that subgoal fails. Finally, the query *s* succeeds, because each of the subgoals *p*, not *q* succeeds. (The latter succeeds because the corresponding positive goal *q* fails.) To sum up, the behavior of SLDNF resolution on the given program can be represented by the following truth assignment:

p	*q*	*r*	*s*
T	**F**	**F**	**T**.

On the other hand, the rules of the given program can be viewed as propositional formulas if we identify the comma with conjunction $\wedge$, the symbol not with negation $\neg$, and agree to treat $F \leftarrow G$ as the implication $G \rightarrow F$ written backwards. For instance, the last rule of the given program is, from this point of view, alternative notation for the propositional formula

$$p \wedge \neg q \rightarrow s.$$

If we calculate the truth values of the rules of the program for the truth assignment shown above then we will see that each rule gets the value **T**. In other words, that assignment is a model of the program. But this program has also other models, for instance

p	*q*	*r*	*s*
T	**T**	**T**	**F**.

Thus one of the models of the given program is special in the sense that it correctly represents the behavior of SLDNF resolution. What are the mathematical properties of that model that make it special? An answer to this question is provided by the definition of a stable model.

Relation to nonmonotonic logic

The meaning of negation in logic programs is closely related to two theories of nonmonotonic reasoning -- autoepistemic logic and default logic. The discovery of these relationships was a key step towards the invention of the stable model semantics.

The syntax of autoepistemic logic uses a modal operator that allows us to distinguish between what is true and what is believed. Michael Gelfond [1987] proposed to read $\text{not } p$ in the body of a rule as "*p* is not believed", and to understand a rule with negation as the corresponding formula of autoepistemic logic. The stable model semantics, in its basic form, can be viewed as a reformulation of this idea that avoids explicit references to autoepistemic logic.

In default logic, a default is similar to an inference rule, except that it includes, besides its premises and conclusion, a list of formulas called justifications. A default can be used to derive its conclusion under the assumption that its justifications are consistent with what is currently believed. Nicole Bidoit and Christine Froidevaux [1987] proposed to treat negated atoms in the bodies of rules as justifications. For instance, the rule

$$s \leftarrow p,\ \text{not } q$$

can be understood as the default that allows us to derive *s* from *p* assuming that $\neg q$ is consistent. The stable model semantics uses the same idea, but it does not explicitly refer to default logic.

Stable models

The definition of a stable model below, reproduced from [Gelfond and Lifschitz, 1988], uses two conventions. First, a truth assignment is identified with the set of atoms that get the value **T**. For instance, the truth assignment

p	*q*	*r*	*s*
T	**F**	**F**	**T**

is identified with the set $\{p,s\}$. This convention allows us to use the set inclusion relation to compare truth assignments with each other. The smallest of all truth assignments $\emptyset$ is the one that makes every atom false; the largest truth assignment makes every atom true.

Second, a logic program with variables is viewed as shorthand for the set of all ground instances of its rules, that is, for the result of substituting variable-free terms for variables in the rules of the program in all possible ways. For instance, the logic programming definition of even numbers

$$even(0)$$
$$even(s(X)) \leftarrow \text{not } even(X)$$

is understood as the result of replacing *X* in this program by the ground terms

$$0,\ s(0),\ s(s(0)), \ldots .$$

in all possible ways. The result is the infinite ground program

$$even(0)$$
$$even(s(0)) \leftarrow \text{not } even(0)$$
$$even(s(s(0))) \leftarrow \text{not } even(s(0))$$
$$\ldots$$

Definition

Let *P* be a set of rules of the form

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

where $A, B_1, \ldots, B_m, C_1, \ldots, C_n$ are ground atoms. If *P* does not contain negation (*n* = 0 in every rule of the program) then, by definition, the only stable model of *P* is its model that is minimal relative to set inclusion. (Any program without negation has exactly one minimal model.) To extend this definition to the case of programs with negation, we need the auxiliary concept of the reduct, defined as follows.

For any set *I* of ground atoms, the *reduct* of *P* relative to *I* is the set of rules without negation obtained from *P* by first dropping every rule such that at least one of the atoms C_i in its body

$$B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

belongs to *I*, and then dropping the parts $\text{not } C_1, \ldots, \text{not } C_n$ from the bodies of all remaining rules.

We say that *I* is a *stable model* of *P* if *I* is the stable model of the reduct of *P* relative to *I*. (Since the reduct does not contain negation, its stable model has been already defined.) As the term "stable model" suggests, every stable model of *P* is a model of *P*.

Example

To illustrate these definitions, let us check that {*p*,*s*} is a stable model of the program

$$p$$
$$r \leftarrow p,\ q$$
$$s \leftarrow p,\ \text{not } q.$$

The reduct of this program relative to {*p*,*s*} is

$$p$$
$$r \leftarrow p,\ q$$
$$s \leftarrow p.$$

(Indeed, since $q \notin \{p, s\}$, the reduct is obtained from the program by dropping the part $\text{not } q$.) The stable model of the reduct is {*p*,*s*}. (Indeed, this set of atoms satisfies every rule of the reduct, and it has no proper subsets with the same property.) Thus after computing the stable model of the reduct we arrived at the same set {*p*,*s*} that we started with. Consequently, that set is a stable model.

Checking in the same way the other 15 sets consisting of the atoms $p,\ q,\ r,\ s$shows that this program has no other stable models. For instance, the reduct of the program relative to {*p*,*q*,*r*} is

$$p$$
$$r \leftarrow p,\ q.$$

The stable model of the reduct is {*p*}, which is different from the set {*p*,*q*,*r*} that we started with.

Programs without a unique stable model

A program with negation may have many stable models or no stable models. For instance, the program

$$p \leftarrow \text{not } q$$
$$q \leftarrow \text{not } p$$

has two stable models {*p*}, {*q*}. The one-rule program

$$p \leftarrow \text{not } p$$

has no stable models.

If we think of the stable model semantics as a description of the behavior of Prolog in the presence of negation then programs without a unique stable model can be judged

unsatisfactory: they do not provide an unambiguous specification for Prolog-style query answering. For instance, the two programs above are not reasonable as Prolog programs -- SLDNF resolution does not terminate on them.

But the use of stable models in answer set programming provides a different perspective on such programs. In that programming paradigm, a given search problem is represented by a logic program so that the stable models of the program correspond to solutions. Then programs with many stable models correspond to problems with many solutions, and programs without stable models correspond to unsolvable problems. For instance, the eight queens puzzle has 92 solutions; to solve it using answer set programming, we encode it by a logic program with 92 stable models. From this point of view, logic programs with exactly one stable model are rather special in answer set programming, like polynomials with exactly one root in algebra.

Properties of the stable model semantics

In this section, as in the definition of a stable model above, by a logic program we mean a set of rules of the form

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

where $A, B_1, \ldots, B_m, C_1, \ldots, C_n$ are ground atoms.

Head atoms: If an atom *A* belongs to a stable model of a logic program *P* then *A* is the head of one of the rules of *P*.

Minimality: Any stable model of a logic program *P* is minimal among the models of *P* relative to set inclusion.

The antichain property: If *I* and *J* are stable models of the same logic program then *I* is not a proper subset of *J*. In other words, the set of stable models of a program is an antichain.

NP-completeness: Testing whether a finite ground logic program has a stable model is NP-complete.

Relation to other theories of negation as failure

Program completion

Any stable model of a finite ground program is not only a model of the program itself, but also a model of its completion [Marek and Subrahmanian, 1989]. The converse, however, is not true. For instance, the completion of the one-rule program

$$p \leftarrow p$$

is the tautology $p \leftrightarrow p$. The model $\emptyset$ of this tautology is stable, but its other model $\{p\}$ is not. François Fages [1994] found a syntactic condition on logic programs that eliminates such counterexamples and guarantees the stability of every model of the program's completion. The programs that satisfy his condition are called *tight*.

Fangzhen Lin and Yuting Zhao [2004] showed how to make the completion of a nontight program stronger so that all its nonstable models will be eliminated. The additional formulas that they add to the completion are called *loop formulas*.

Well-founded semantics

The well-founded model of a logic program partitions all ground atoms into three sets: true, false and unknown. If an atom is true in the well-founded model of *P* then it belongs to every stable model of *P*. The converse, generally, does not hold. For instance, the program

$$\begin{aligned} p &\leftarrow \text{not } q \\ q &\leftarrow \text{not } p \\ r &\leftarrow p \\ r &\leftarrow q \end{aligned}$$

has two stable models, {*p*,*r*} and {*q*,*r*}. Even though *r* belongs to both of them, its value in the well-founded model is *unknown*.

Furthermore, if an atom is false in the well-founded model of a program then it does not belong to any of its stable models. Thus the well-founded model of a logic program provides a lower bound on the intersection of its stable models and an upper bound on their union.

Strong negation

Representing incomplete information

From the perspective of knowledge representation, a set of ground atoms can be thought of as a description of a complete state of knowledge: the atoms that belong to the set are known to be true, and the atoms that do not belong to the set are known to be false. A possibly *incomplete* state of knowledge can be described using a consistent but possibly incomplete set of literals; if an atom *p* does not belong to the set and its negation does not belong to the set either then it is not known whether *p* is true.

In the context of logic programming, this idea leads to the need to distinguish between two kinds of negation -- *negation as failure*, discussed above, and *strong negation*, which is denoted here by ˜. The following example, illustrating the difference between the two kinds of negation, belongs to John McCarthy. A school bus may cross railway tracks under the condition that there is no approaching train. If we do not necessarily know whether a train is approaching then the rule using negation as failure

$$\text{Cross} \leftarrow \text{not Train}$$

is not an adequate representation of this idea: it says that it's okay to cross *in the absence of information* about an approaching train. The weaker rule, that uses strong negation in the body, is preferable:

$$\text{Cross} \leftarrow \sim \text{Train}.$$

It says that it's okay to cross if we *know* that no train is approaching.

Coherent stable models

To incorporate strong negation in the theory of stable models, Gelfond and Lifschitz [1991] allowed each of the expressions A, B_i, C_i in a rule

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

to be either an atom or an atom prefixed with the strong negation symbol. Instead of stable models, this generalization uses *answer sets*, which may include both atoms and atoms prefixed with strong negation.

An alternative approach [Ferraris and Lifschitz, 2005] treats strong negation as a part of an atom, and it does not require any changes in the definition of a stable model. In this theory of strong negation, we distinguish between atoms of two kinds, *positive* and *negative*, and assume that each negative atom is an expression of the form $\sim A$, where $A$is a positive atom. A set of atoms is called *coherent* if it does not contain "complementary" pairs of atoms $A, \sim A$. Coherent stable models of a program are identical to its consistent answer sets in the sense of [Gelfond and Lifschitz, 1991].

For instance, the program

$$\begin{array}{l} p \leftarrow \text{not } q \\ q \leftarrow \text{not } p \\ r \\ \sim r \leftarrow \text{not } p \end{array}$$

has two stable models, $\{p, r\}$and $\{q, r, \sim r\}$. The first model is coherent; the second is not, because it contains both the atom $r$and the atom $\sim r$.

Closed world assumption

According to [Gelfond and Lifschitz, 1991], the closed world assumption for a predicate $p$can be expressed by the rule

$$\sim p(X_1, \ldots, X_n) \leftarrow \text{not } p(X_1, \ldots, X_n)$$

(the relation p does not hold for a tuple $X_1, \ldots, X_n$ if there is no evidence that it does). For instance, the stable model of the program

$$p(a, b)$$
$$p(c, d)$$
$$\sim p(X, Y) \leftarrow \text{not } p(X, Y)$$

consists of 2 positive atoms

$$p(a, b),\ p(c, d)$$

and 14 negative atoms

$$\sim p(a, a),\ \sim (a, c),\ \ldots$$

-- the strong negations of all other positive ground atoms formed from $p,\ a,\ b,\ c,\ d$.

A logic program with strong negation can include the closed world assumption rules for some of its predicates and leave the other predicates in the realm of the open world assumption.

Programs with constraints

The stable model semantics has been generalized to many kinds of logic programs other than collections of "traditional" rules discussed above -- rules of the form

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

where $A, B_1, \ldots, B_m, C_1, \ldots, C_n$ are atoms. One simple extension allows programs to contain *constraints* -- rules with the empty head:

$$\leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n.$$

Recall that a traditional rule can be viewed as alternative notation for a propositional formula if we identify the comma with conjunction $\wedge$, the symbol not with negation $\neg$, and agree to treat $F \leftarrow G$ as the implication $G \rightarrow F$ written backwards. To extend this convention to constraints, we identify a constraint with the negation of the formula corresponding to its body:

$$\neg(B_1 \wedge \cdots \wedge B_m \wedge \neg C_1 \wedge \cdots \wedge \neg C_n).$$

We can now extend the definition of a stable model to programs with constraints. As in the case of traditional programs, we begin with programs that do not contain negation. Such a program may be inconsistent; then we say that it has no stable models. If such a

program *P* is consistent then *P* has a unique minimal model, and that model is considered the only stable model of *P*.

Next, stable models of arbitrary programs with constraints are defined using reducts, formed in the same way as in the case of traditional programs. A set *I* of atoms is a *stable model* of a program *P* with constraints if the reduct of *P* relative to *I* has a stable model, and that stable model equals *I*.

The properties of the stable model semantics stated above for traditional programs hold in the presence of constraints as well.

Constraints play an important role in answer set programming because adding a constraint to a logic program *P* affects the collection of stable models of *P* in a very simple way: it eliminates the stable models that violate the constraint. In other words, for any program *P* with constraints and any constraint *C*, the stable models of $P \cup \{C\}$ can be characterized as the stable models of *P* that satisfy *C*.

Disjunctive programs

In a *disjunctive rule*, the head may be the disjunction of several atoms:

$$A_1; \ldots; A_k \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n$$

(the semicolon is viewed as alternative notation for disjunction $\vee$). Traditional rules correspond to $k = 1$, and constraints to $k = 0$. To extend the stable model semantics to disjunctive programs [Gelfond and Lifschitz, 1991], we first define that in the absence of negation ($n = 0$ in each rule) the stable models of a program are its minimal models. The definition of the reduct for disjunctive programs remains the same as before. A set *I* of atoms is a *stable model* of *P* if *I* is a stable model of the reduct of *P* relative to *I*.

For example, the set $\{p,r\}$ is a stable model of the disjunctive program

$$\begin{array}{l} p; q \\ r \leftarrow \text{not } q \end{array}$$

because it is one of two minimal models of the reduct

$$\begin{array}{l} p; q \\ r. \end{array}$$

The program above has one more stable model, $\{q\}$.

As in the case of traditional programs, each element of any stable model of a disjunctive program *P* is a head atom of *P*, in the sense that it occurs in the head of one of the rules of *P*. As in the traditional case, the stable models of a disjunctive program are minimal

and form an antichain. Testing whether a finite disjunctive program has a stable model is Σ_2^{P}-complete [Eiter and Gottlob, 1993].

Stable models of a set of propositional formulas

Rules, and even disjunctive rules, have a rather special syntactic form, in comparison with arbitrary propositional formulas. Each disjunctive rule is essentially an implication such that its antecedent (the body of the rule) is a conjunction of literals, and its consequent (head) is a disjunction of atoms. David Pearce [1997] and Paolo Ferraris [2005] showed how to extend the definition of a stable model to sets of arbitrary propositional formulas. This generalization has applications to answer set programming.

Pearce's formulation looks very different from the original definition of a stable model. Instead of reducts, it refers to *equilibrium logic* -- a system of nonmonotonic logic based on Kripke models. Ferraris's formulation, on the other hand, is based on reducts, although the process of constructing the reduct that it uses differs from the one described above. The two approaches to defining stable models for sets of propositional formulas are equivalent to each other.

General definition of a stable model

According to [Ferraris, 2005], the *reduct* of a propositional formula *F* relative to a set *I* of atoms is the formula obtained from *F* by replacing each maximal subformula that is not satisfied by *I* with the logical constant $\bot$(false). The *reduct* of a set *P* of propositional formulas relative to *I* consists of the reducts of all formulas from *P* relative to *I*. As in the case of disjunctive programs, we say that a set *I* of atoms is a *stable model* of *P* if *I* is minimal (with respect to set inclusion) among the models of the reduct of *P* relative to *I*.

For instance, the reduct of the set

$$\{p,\ p \wedge q \rightarrow r,\ p \wedge \neg q \rightarrow s\}$$

relative to {*p*,*s*} is

$$\{p,\ \bot \rightarrow \bot,\ p \wedge \neg\bot \rightarrow s\}.$$

Since {*p*,*s*} is a model of the reduct, and the proper subsets of that set are not models of the reduct, {*p*,*s*} is a stable model of the given set of formulas.

We have seen that {*p*,*s*} is also a stable model of the same formula, written in logic programming notation, in the sense of the original definition. This is an instance of a general fact: in application to a set of (formulas corresponding to) traditional rules, the definition of a stable model according to Ferraris is equivalent to the original definition. The same is true, more generally, for programs with constraints and for disjunctive programs.

Properties of the general stable model semantics

The theorem asserting that all elements of any stable model of a program P are head atoms of P can be extended to sets of propositional formulas, if we define head atoms as follows. An atom A is a *head atom* of a set P of propositional formulas if at least one occurrence of A in a formula from P is neither in the scope of a negation nor in the antecedent of an implication. (We assume here that equivalence is treated as an abbreviation, not a primitive connective.)

The minimality and the antichain property of stable models of a traditional program do not hold in the general case. For instance, (the singleton set consisting of) the formula

$$p \vee \neg p$$

has two stable models, $\emptyset$ and $\{p\}$. The latter is not minimal, and it is a proper superset of the former.

Testing whether a finite set of propositional formulas has a stable model is Σ_2^{P}-complete, as in the case of disjunctive programs.

Chapter 4

Answer Set Programming and Abductive Logic Programming

Answer set programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems. It is based on the stable model (answer set) semantics of logic programming. In ASP, search problems are reduced to computing stable models, and *answer set solvers* -- programs for generating stable models -- are used to perform search. The computational process employed in the design of many answer set solvers is an enhancement of the DPLL algorithm and, in principle, it always terminates (unlike Prolog query evaluation, which may lead to an infinite loop).

In a more general sense, ASP includes all applications of answer sets to knowledge representation and the use of Prolog-style query evaluation for solving problems arising in these applications.

History

The planning method proposed by Dimopoulos, Nebel and Köhler is an early example of answer set programming. Their approach is based on the relationship between plans and stable models. Soininen and Niemelä applied what is now known as answer set programming to the problem of product configuration. The use of answer set solvers for search was identified as a new programming paradigm in Marek and Truszczyński (the term "answer set programming" was used for the first time as the title of a part of the collection where that paper appeared) and in [Niemelä 1999].

Answer set programming language Lparse

Lparse is the name of the program that was originally created as a front-end for the answer set solver smodels, and is now used in the same way in many other answer set solvers, including assat, clasp, cmodels, gNt, nomore++ and pbmodels. (dlv is an exception; the syntax of ASP programs written for dlv is somewhat different.)

An Lparse program consists of rules of the form

```
<head> :- <body> .
```

The symbol `:-` ("if") is dropped if `<body>` is empty. The simplest kind of Lparse rules are rules with constraints.

One other useful construct included in this language is *choice*. For instance, the choice rule

```
{p,q,r}.
```

says: choose arbitrarily which of the atoms *p*,*q*,*r* to include in the stable model. The lparse program that contains this choice rule and no other rules has 8 stable models -- arbitrary subsets of {*p*,*q*,*r*}. The definition of a stable model was generalized to programs with choice rules. Choice rules can be treated also as abbreviations for propositional formulas under the stable model semantics. For instance, the choice rule above can be viewed as shorthand for the conjunction of three "excluded middle" formulas:

$$(p \vee \neg p) \wedge (q \vee \neg q) \wedge (r \vee \neg r).$$

The language of lparse allows us also to write "constrained" choice rules, such as

```
1{p,q,r}2.
```

This rule says: choose at least 1 of the atoms *p*,*q*,*r*, but not more than 2. The meaning of this rule under the stable model semantics is represented by the propositional formula

$$(p \vee \neg p) \wedge (q \vee \neg q) \wedge (r \vee \neg r) \\ \wedge (p \vee q \vee r) \wedge \neg(p \wedge q \wedge r).$$

Cardinality bounds can be used in the body of a rule as well, for instance:

```
:- 2{p,q,r}.
```

Adding this constraint to an Lparse program eliminates the stable models that contain at least 2 of the atoms *p*,*q*,*r*. The meaning of this rule can be represented by the propositional formula

$$\neg((p \wedge q) \vee (p \wedge r) \vee (q \wedge r)).$$

Variables (capitalized, as in Prolog) are used in Lparse to abbreviate collections of rules that follow the same pattern, and also to abbreviate collections of atoms within the same rule. For instance, the Lparse program

```
p(a). p(b). p(c).
q(X) :- p(X), X!=a.
```

has the same meaning as

```
p(a). p(b). p(c).
q(b). q(c).
```

The program

```
p(a). p(b). p(c).
{q(X):p(X)}2.
```

is shorthand for

```
p(a). p(b). p(c).
{q(a),q(b),q(c)}2.
```

Generating stable models

To find a stable model of the Lparse program stored in file `<filename>` we use the command

```
% lparse <filename> | smodels
```

Option 0 instructs smodels to find *all* stable models of the program. For instance, if file `test` contains the rules

```
1{p,q,r}2.
s :- not p.
```

then the command

```
% lparse test | smodels 0
```

produces the output

```
Answer: 1
Stable Model: q p
Answer: 2
Stable Model: p
Answer: 3
Stable Model: r p
Answer: 4
Stable Model: q s
Answer: 5
Stable Model: r s
Answer: 6
Stable Model: r q s
```

Examples of ASP programs

Graph coloring

An *n*-coloring of a graph *G* is a function $color$ from its set of vertices to $\{1, \ldots, n\}$ such that $color(x) \neq color(y)$ for every pair of adjacent vertices *x*,*y*. We would like to use ASP to find an *n*-coloring of a given graph (or determine that it does not exist).

This can be accomplished using the following Lparse program:

```
c(1..n).
1 {color(X,I) : c(I)} 1 :- v(X).
:- color(X,I), color(Y,I), e(X,Y), c(I).
```

Line 1 defines the numbers $1, \ldots, n$ to be colors. According to the choice rule in Line 2, a unique color *i* should be assigned to each vertex *x*. The constraint in Line 3 prohibits assigning the same color to vertices *x* and *y* if there is an edge connecting them.

If we combine this file with a definition of *G*, such as

```
v(1..100). % 1,...,100 are vertices
e(1,55). % there is an edge from 1 to 55
. . .
```

and run smodels on it, with the numeric value of *n* specified on the command line, then the atoms of the form $color(\ldots, \ldots)$ in the output of smodels will represent an *n*-coloring of *G*.

The program in this example illustrates the "generate-and-test" organization that is often found in simple ASP programs. The choice rule describes a set of "potential solutions" -- a simple superset of the set of solutions to the given search problem. It is followed by a constraint, which eliminates all potential solutions that are not acceptable. However, the search process employed by smodels and other answer set solvers is not based on trial and error.

Large clique

A clique in a graph is a set of pairwise adjacent vertices. The following lparse program finds a clique of size $\geq n$ in a given graph, or determines that it does not exist:

```
n {in(X) : v(X)}.
:- in(X), in(Y), v(X), v(Y), X!=Y, not e(X,Y), not e(Y,X).
```

This is another example of the generate-and-test organization. The choice rule in Line 1 "generates" all sets consisting of $\geq n$ vertices. The constraint in Line 2 "weeds out" the sets that are not cliques.

Hamiltonian cycle

A Hamiltonian cycle in a directed graph is a cycle that passes through each vertex of the graph exactly once. The following Lparse program can be used to find a Hamiltonian cycle in a given directed graph if it exists; we assume that 0 is one of the vertices.

```
{in(X,Y)} :- e(X,Y).

:- 2 {in(X,Y) : e(X,Y)}, v(X).
:- 2 {in(X,Y) : e(X,Y)}, v(Y).

r(X) :- in(0,X), v(X).
r(Y) :- r(X), in(X,Y), e(X,Y).

:- not r(X), v(X).
```

The choice rule in Line 1 "generates" all subsets of the set of edges. The three constraints "weed out" the subsets that are not Hamiltonian cycles. The last of them uses the auxiliary predicate $r(x)$ ("x is reachable from 0") to prohibit the vertices that do not satisfy this condition. This predicate is defined recursively in Lines 4 and 5.

This program is an example of the more general "generate, define and test" organization: it includes the definition of an auxiliary predicate that helps us eliminate all "bad" potential solutions.

Comparison of implementations

	Platform		Features			Mechanics
Name	**OS**	**Licence**	**Variables**	**Function symbols**	**Explicit sets**	**Explicit lists**
DLV	Linux, Mac OS, Windows	Freeware	Yes	No	No	No
Smodels	Linux, Mac OS, Windows	GPL	Yes	No	No	No

Abductive logic programming

Abductive logic programming (**ALP**) is a high level knowledge-representation framework that can be used to solve problems declaratively based on abductive reasoning. It extends normal logic programming by allowing some predicates to be incompletely defined, declared as abducible predicates. Problem solving is effected by deriving hypotheses on these abducible predicates (abductive hypotheses) as solutions of problems to be solved. These problems can be either observations that need to be explained (as in classical abduction) or goals to be achieved (as in normal logic

programming). It can be used to solve problems in Diagnosis, Planning, Natural Language and Machine Learning. It has also been used to interpret Negation as failure as a form of abductive reasoning.

Syntax

Abductive logic programs have three components, $< P,A,IC >$, where:

- P is a logic program of exactly the same form as in Logic Programming
- A is a set of predicate names, called the abducible predicates
- IC is a set of first order classical formulae.

Normally, the logic program P does not contain any clauses whose head (or conclusion) refers to an abducible predicate. (This restriction can be made without loss of generality.) Also in practice, many times the integrity constraints in IC are often restricted to the form of denials, i.e. clauses of the form:

```
false:- A1,...,An, not B1, ..., not Bm.
```

Such a constraint means that it is not possible for all A1,...,An to be true and at the same time all of B1,...,Bm to be false.

Informal meaning and problem solving

The clauses in P define a set of non-abducible predicates and through this they provide a description (or model) of the problem domain. The integrity constraints in IC specify general properties of the problem domain that need to be respected in any solution of our problem.

A problem, G, which expresses either an observation that needs to be explained or a goal that is desired, is represented by a conjunction of positive and negative (NAF) literals. Such problems are solved by computing "abductive explanations" of G.

An abductive explanation of a problem G is a set of positive (and sometimes also negative) ground instances of the abducible predicates, such that, when these are added to the logic program P, the problem G and the integrity constraints IC both hold. Thus abductive explanations extend the logic program P by the addition of full or partial definitions of the abducible predicates. In this way, abductive explanations form solutions of the problem according to the description of the problem domain in P and IC. The extension or completion of the problem description given by the abductive explanations provides new information, hitherto not contained in the solution to the problem. Quality criteria to prefer one solution over another, often expressed via integrity constraints, can be applied to select specific abductive explanations of the problem G.

Computation in ALP combines the backwards reasoning of normal logic programming (to reduce problems to sub-problems) with a kind of integrity checking to show that the abductive explanations satisfy the integrity constraints.

The following two examples, written in simple structured English rather than in the strict syntax of ALP, illustrate the notion of abductive explanation in ALP and its relation to problem solving.

Example 1

The abductive logic program, < *P*,*A*,*IC* > , has in *P* the following sentences:

```
Grass is wet if it rained.
Grass is wet if the sprinkler was on.
The sun was shining.
```

The abducible predicates in *A* are "it rained" and "the sprinkler was on" and the only integrity constraintin in *IC*is:

```
false if it rained and the sun was shining.
```

The observation that the grass is wet has two potential explanations, "it rained" and "the sprinkler was on", which entail the observation. However, only the second potential explanation, "the sprinkler was on", satisfies the integrity constraint.

Example 2

Consider the abductive logic program consisting of the following (simplified) clauses:

```
X is citizen if X is born in USA.
X is citizen if X is born outside USA and X is resident of USA and X
is naturalized.
X is citizen if X is born outside USA and Y is mother of X and Y is
citizen and X is registered.
Mary is mother of John.
Mary is citizen.
```

together with the five abducible predicates, "is born in USA", "is born outside USA", "is resident of USA", "is naturalized" and "is registered" and the integrity constraint:

```
false if John is resident of USA.
```

The goal "John is citizen" has two abductive solutions, one of which is "John is born in USA", the other of which is "John is born outside USA" and "John is registered". The potential solution of becoming a citizen by residence and naturalization fails because it violates the integrity constraint.

A more complex example that is also written in the more formal syntax of ALP is the following.

Example 3

The abductive logic program below describes a simple model of the lactose metabolism of the bacterium E. Coli. The program P describes the fact that E. coli can feed on the sugar lactose if it makes two enzymes permease and galactosidase. Like all enzymes (E), these are made if they are coded by a gene (G) that is expressed. These enzymes are coded by two genes (lac(y) and lac(z)) in cluster of genes (lac(X)) – called an operon – that is expressed when the amounts (amt) of glucose are low and lactose are high or when they are both at medium level. The abducibles, A, declare all ground instances of the predicates "amount" as assumable. This reflects the fact that in the model the amounts at any time of the various substances are unknown. This is incomplete information that is to be determined in each problem case. The integrity constraints state that the amount of a substance (S) can only take one value.

Domain Knowledge (P)

```
feed(lactose):-make(permease),make(galactosidase).
make(Enzyme):-code(Gene,Enzyme),express(Gene).
express(lac(X)):-amount(glucose,low),amount(lactose,hi).
express(lac(X)):-amount(glucose,medium),amount(lactose,medium).
code(lac(y),permease).
code(lac(z),galactosidase).
temperature(low):-amount(glucose,low).
```

Integrity Constraints (IC)

```
false :- amount(S,V1), amount(S,V2), V1 ≠ V2.
```

Abducibles (A)

```
abducible_predicate(amount).
```

The problem goal is $G = feed(lactose)$. This can arise either as an observation to be explained or as a state of affairs to be achieved by finding a plan. This goal has two abductive explanations:

$\Delta_1 = \{amount(lactose,hi), amount(glucose,low)\}$

$\Delta_2 = \{amount(lactose,medium), amount(glucose,medium)\}$

The decision which of the two to adopt could depend on addition information that is available, e.g. it may be known that when the level of glucose is low then the organism

exhibits a certain behaviour - in the model such additional information is that the temperature of the organism is low - and by observing the truth or falsity of this it is possible to choose the first or second explanation respectively.

Once an explanation has been chosen, then this becomes part of the theory, which can be used to draw new conclusions. The explanation and more generally these new conclusions form the solution of the problem.

Formal semantics

The formal semantics of the central notion of an abductive explanation in ALP, can be defined in the following way:

Given an abductive logic program, < *P*,*A*,*IC* > , an abductive explanation for a problem *G* is a set Δ of ground atoms on abducible predicates such that:

- $P \cup \Delta \models G$
- $P \cup \Delta \models IC$
- $P \cup \Delta \; is \; [[consistent]].$

This definition is generic in the underlying semantics of logic programming. Each particular choice of semantics defines its own entailment relation $\models$, its own notion of consistent logic programs and hence its own notion of what an abductive solution is. In practice, the three main semantics of logic programming --- completion, stable and well-founded semantics --- have been used to define different ALP frameworks.

The integrity constraints *IC* define *how* they constrain the abductive solutions. There are different views on this. Early work on abduction in Theorist in the context of classical logic was based on the *consistency view* on constraints. In this view, any extension of the given theory with an abductive solution Δ is required to be consistent with the integrity constraints IC: $P \cup IC \cup \Delta$is consistent. The above definition formalizes the *entailment view*: the abductive solution Δ together with *P* should entail the constraints. This view is the one taken in most versions of ALP and is stronger than the consistency view in the sense that a solution according to the entailment view is a solution according to the consistency view but not vice versa.

The difference between the two views can be subtle but in practice the different views usually coincide. E.g. it frequently happens that $P \cup \Delta$has a unique model, in which case the two views are equivalent. In practice, many ALP systems use the entailment view as this can be easily implemented without the need for any extra specialized procedures for the satisfaction of the integrity constraints since this semantics treats the constraints in the same way as the goal.

Comparison with other frameworks

ALP can be viewed as a refinement of Theorist, which explored the use of abduction in first-order logic for both explanation and default reasoning. David Poole later developed a logic programming variant of Theorist, in which abducible predicates have associated probabilities. He showed that probabilistic Horn abduction incorporates both pure Prolog and Bayesian networks as special cases.

There exist strong links between some ALP frameworks and other extensions of logic programming. In particular, ALP has close connections with Answer Set Programming. An abductive logic program can be translated into an equivalent answer set program under the stable model semantics. Consequently, systems for computing stable models such as SMODELS can be used to compute abduction in ALP.

ALP is also closely related to constraint logic programming (CLP). On the one hand, the integration of constraint solving and abductive logic programming enhances the practical utility of ALP through a more efficient computation of abduction. On the other hand, the integration of ALP and CLP can be seen as a high-level constraint programming environment that allows more modular and flexible representations of the problem domain.

There is also a strong link between ALP and Argumentation in logic programming. This relates both to the interpretation of Negation as Failure and Integrity Constraints.

Implementation and systems

Most of the implementations of ALP extend the SLD resolution based computational model of logic programming. ALP can also be implemented by means on its link with Answer Set Programming (ASP), where the ASP systems can be employed. Examples of systems of the former approach are ACLP, A-system, CIFF, SCIFF, ABDUAL and ProLogICA.

Chapter 5

Constraint Logic Programming

Constraint logic programming is a form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction. A constraint logic program is a logic program that contains constraints in the body of clauses. An example of a clause including a constraint is `A(X,Y) :- X+Y>0, B(X), C(Y)`. In this clause, `X+Y>0` is a constraint; `A(X,Y)`, `B(X)`, and `C(Y)` are literals as in regular logic programming. This clause states one condition under which the statement `A(X,Y)` holds: `X+Y` is greater than zero and both `B(X)` and `C(Y)` are true.

As in regular logic programming, programs are queried about the provability of a goal, which may contain constraints in addition to literals. A proof for a goal is composed of clauses whose bodies are satisfiable constraints and literals that can in turn be proved using other clauses. Execution is performed by an interpreter, which starts from the goal and recursively scans the clauses trying to prove the goal. Constraints encountered during this scan are placed in a set called **constraint store**. If this set is found out to be unsatisfiable, the interpreter backtracks, trying to use other clauses for proving the goal. In practice, satisfiability of the constraint store may be checked using an incomplete algorithm, which does not always detect inconsistency.

Overview

Formally, constraint logic programs are like regular logic programs, but the body of clauses can contain constraints, in addition to the regular logic programming literals. As an example, `X>0` is a constraint, and is included in the last clause of the following constraint logic program.

```
B(X,1):- X<0.
B(X,Y):- X=1, Y>0.
A(X,Y):- X>0, B(X,Y).
```

Like in regular logic programming, evaluating a goal such as `A(X,1)` requires evaluating the body of the last clause with `Y=1`. Like in regular logic programming, this in turn requires proving the goal `B(X,1)`. Contrary to regular logic programming, this also requires a constraint to be satisfied: `X>0`, the constraint in the body of the last clause.

Whether a constraint is satisfied cannot always be determined when the constraint is encountered. In this case, for example, the value of `X` is not determined when the last clause is evaluated. As a result, the constraint `X>0` is not satisfied nor violated at this point. Rather than proceeding in the evaluation of `B(X,1)` and then checking whether the resulting value of `X` is positive afterwards, the interpreter stores the constraint `X>0` and then proceeds in the evaluation of `B(X,1)`; this way, the interpreter can detect violation of the constraint `X>0` during the evaluation of `B(X,1)`, and backtrack immediately if this is the case, rather than waiting for the evaluation of `B(X,1)` to conclude.

In general, the evaluation of a constraint logic program proceeds like for a regular logic program, but constraint encountered during evaluation are placed in a set called constraint store. As an example, the evaluation of the goal `A(X,1)` proceeds by evaluating the body of the first clause with `Y=1`; this evaluation adds `X>0` to the constraint store and requires the goal `B(X,1)` to be proved. While trying to prove this goal, the first clause is applicable, but its evaluation adds `X<0` to the constraint store. This addition makes the constraint store unsatisfiable, and the interpreter backtracks, removing the last addition from the constraint store. The evaluation of the second clause adds `X=1` and `Y>0` to the constraint store. Since the constraint store is satisfiable and no other literal is left to prove, the interpreter stops with the solution `X=1, Y=1`.

Semantics

The semantics of constraint logic programs can be defined in terms of a virtual interpreter that maintains a pair $\langle G, S \rangle$during execution. The first element of this pair is called current goal; the second element is called constraint store. The *current goal* contains the literals the interpreter is trying to prove and may also contain some constraints it is trying to satisfy; the *constraint store* contains all constraints the interpreter has assumed satisfiable so far.

Initially, the current goal is the goal and the constraint store is empty. The interpreter proceeds by removing the first element from the current goal and analyzing it. The details of this analysis are explained below, but in the end this analysis may produce a successful termination or a failure. This analysis may involve recursive calls and addition of new literals to the current goal and new constraint to the constraint store. The interpreter backtracks if a failure is generated. A successful termination is generated when the current goal is empty and the constraint store is satisfiable.

The details of the analysis of a literal removed from the goal is as follows. After having removed this literal from the front of the goal, it is checked whether it is a constraint or a literal. If it is a constraint, it is added to the constraint store. If it is a literal, a clause whose head has the same predicate of the literal is chosen; the clause is rewritten by replacing its variables with new variables (variables not occurring in the goal): the result is called a *fresh variant* of the clause; the body of the fresh variant of the clause is then placed in front of the goal; the equality of each argument of the literal with the corresponding one of the fresh variant head is placed in front of the goal as well.

Some checks are done during these operations. In particular, the constraint store is checked for consistency every time a new constraint is added to it. In principle, whenever the constraint store is unsatisfiable the algorithm could backtrack. However, checking unsatisfiability at each step would be inefficient. For this reason, an incomplete satisfiability checker may be used instead. In practice, satisfiability is checked using methods that simplify the constraint store, that is, rewrite it into an equivalent but simpler-to-solve form. These methods can sometimes but not always prove unsatisfiability of an unsatisfiable constraint store.

The interpreter has proved the goal when the current goal is empty and the constraint store is not detected unsatisfiable. The result of execution is the current set of (simplified) constraints. This set may include constraints such as $X = 2$ that force variables to a specific value, but may also include constraints like $X > 2$ that only bound variables without giving them a specific value.

Formally, the semantics of constraint logic programming is defined in terms of *derivations*. A transition is a pair of pairs goal/store, noted $\langle G, S\rangle \rightarrow \langle G', S'\rangle$. Such a pair states the possibility of going from state $\langle G, S\rangle$ to state $\langle G', S'\rangle$. Such a transition is possible in three possible cases:

- an element of *G* is a constraint *C*, $G' = G \backslash \{C\}$ and $S' = S \cup \{C\}$; in other words, a constraint can be moved from the goal to the constraint store
- an element of *G* is a literal $L(t_1, \ldots, t_n)$, there exists a clause that, rewritten using new variables, is $L(t'_1, \ldots, t'_n) :- B$, *G'* is *G* with $L(t_1, \ldots, t_n)$ replaced by $t_1 = t'_1, \ldots, t_n = t'_n, B$, and *S'* = *S*; in other words, a literal can be replaced by the body of a fresh variant of a clause having the same predicate in the head, adding the body of the fresh variant and the above equalities of terms to the goal
- *S* and *S'* are equivalent according to the specific constraint semantics

A sequence of transitions is a derivation. A goal *G* can be proved if there exists a derivation from $\langle G, \emptyset\rangle$ to $\langle \emptyset, S\rangle$ for some satisfiable constraint store *S*. This semantics formalizes the possible evolutions of an interpreter that arbitrarily chooses the literal of the goal to process and the clause to replace literals. In other words, a goal is proved under this semantics if there exists a sequence of choices of literals and clauses, among the possibly many ones, that lead to an empty goal and satisfiable store.

Actual interpreters process the goal elements in a LIFO order: elements are added in the front and processed from the front. They also choose the clause of the second rule according to the order in which they are written, and rewrite the constraint store when it is modified.

The third possible kind of transition is a replacement of the constraint store with an equivalent one. This replacement is limited to those done by specific methods, such as constraint propagation. The semantics of constraint logic programming is parametric not only to the kind of constraints used but also to the method for rewriting the constraint store. The specific methods used in practice replace the constraint store with one that is simpler to solve. If the constraint store is unsatisfiable, this simplification may detect this unsatisfiability sometimes, but not always.

The result of evaluating a goal against a constraint logic program is defined if the goal is proved. In this case, there exists a derivation from the initial pair to a pair where the goal is empty. The constraint store of this second pair is considered the result of the evaluation. This is because the constraint store contains all constraints assumed satisfiable to prove the goal. In other words, the goal is proved for all variable evaluations that satisfy these constraints.

The pairwise equality of terms of two literals is often compactly denoted by $L(t_1, \ldots, t_n) = L(t'_1, \ldots, t'_n)$: this is a shorthand for the constraints $t_1 = t'_1, \ldots, t_n = t'_n$. A common variant of the semantics for constraint logic programming adds $L(t_1, \ldots, t_n) = L(t'_1, \ldots, t'_n)$ directly to the constraint store rather than to the goal.

Terms and constraints

Different definitions of terms are used, generating different kinds of constraint logic programming: over trees, reals, or finite domains. A kind of constraint that is always present is the equality of terms. Such constraints are necessary because the interpreter adds `t1=t2` to the goal whenever a literal `P(...t1...)` is replaced with the body of a clause fresh variant whose head is `P(...t2...)`.

Tree terms

Constraint logic programming with tree terms emulates regular logic programming by storing substitutions as constraints in the constraint store. Terms are variables, constants, and function symbols applied to other terms. The only considered constraints are equalities and disequalities between terms. Equality is particularly important, as constraints link `t1=t2` are often generated by the interpreter. Equality constraints on terms can be simplified, that is solved, via unification:

A constraint `t1=t2` can be simplified if both terms are function symbols applied to other terms. If the two function symbols are the same and the number of subterms is also the same, this constraint can be replaced with the pairwise equality of subterms. If the terms are composed of different function symbols or the same functor but on different number of terms, the constraint is unsatisfiable.

If one of the two terms is a variable, the only allowed value the variable can take is the other term. As a result, the other term can replace the variable in the current goal and constraint store, thus practically removing the variable from consideration. In the particular case of equality of a variable with itself, the constraint can be removed as always satisfied.

In this form of constraint satisfaction, variable values are terms.

Reals

Constraint logic programming with real numbers uses real expressions as terms. When no function symbols are used, terms are expressions over reals, possibly including variables. In this case, each variable can only take a real number as a value.

To be precise, terms are expressions over variables and real constants. Equality between terms is a kind of constraint that is always present, as the interpreter generates equality of terms during execution. As an example, if the first literal of the current goal is `A(X+1)` and the interpreter has chosen a clause that is `A(Y-1):-Y=1` after rewriting is variables, the constraints added to the current goal are `X+1=Y-1` and $Y = 1$. The rules of simplification used for function symbols are obviously not used: `X+1=Y-1` is not unsatisfiable just because the first expression is built using `+` and the second using `-`.

Reals and function symbols can be combined, leading to terms that are expressions over reals and function symbols applied to other terms. Formally, variables and real constants are expressions, as any arithmetic operator over other expressions. Variables, constants (zero-arity-function symbols), and expressions are terms, as any function symbol applied to terms. In other words, terms are built over expressions, while expressions are built over numbers and variables. In this case, variables ranges over real numbers *and terms*. In other words, a variable can take a real number as a value, while another takes a term.

Equality of two terms can be simplified using the rules for tree terms if none of the two terms is a real expression. For example, if the two terms have the same function symbol and number of subterms, their equality constraint can be replaced with the equality of subterms.

Finite domains

The third class of constraints used in constraint logic programming is that of finite domains. Values of variables are in this case taken from a finite domain, often that of integer numbers. For each variable, a different domain can be specified: `X::[1..5]` for example means that the value of `X` is between `1` and `5`. The domain of a variable can also be given by enumerating all values a variable can take; therefore, the above domain declaration can be also written `X::[1,2,3,4,5]`. This second way of specifying a domain allows for domains that are not composed of integers, such as `X::[george,mary,john]`. If the domain of a variable is not specified, it is assumed to be

the set of integers representable in the language. A group of variables can be given the same domain using a declaration like `[X,Y,Z]::[1..5]`.

The domain of a variable may be reduced during execution. Indeed, as the interpreter adds constraints to the constraint store, it performs constraint propagation to enforce a form of local consistency, and these operations may reduce the domain of variables. If the domain of a variable becomes empty, the constraint store is inconsistent, and the algorithm backtracks. If the domain of a variable becomes a singleton, the variable can be assigned the unique value in its domain. The forms of consistency typically enforced are arc consistency, hyper-arc consistency, and bound consistency. The current domain of a variable can be inspected using specific literals; for example, `dom(X,D)` finds out the current domain `D` of a variable `X`.

As for domains of reals, functors can be used with domains of integers. In this case, a term can be an expression over integers, a constant, or the application of a functor over other terms. A variable can take an arbitrary term as a value, if its domain has not been specified to be a set of integers or constants.

The constraint store

The constraint store contains the constraints that are currently assumed satisfiable. It can be considered what the current substitution is for regular logic programming. When only tree terms are allowed, the constraint store contains constraints in the form `t1=t2`; these constraints are simplified by unification, resulting in constraints of the form `variable=term`; such constraints are equivalent to a substitution.

However, the constraint store may also contain constraints in the form `t1!=t2`, if the difference `!=` between terms is allowed. When constraints over reals or finite domains are allowed, the constraint store may also contain domain-specific constraints like `X+2=Y/2`, etc.

The constraint store extends the concept of current substitution in two ways. First, it does not only contain the constraints derived from equating a literal with the head of a fresh variant of a clause, but also the constraints of the body of clauses. Second, it does not only contain constraints of the form `variable=value` but also constraints on the considered constraint language. While the result of a successful evaluation of a regular logic program is the final substitution, the result for a constraint logic program is the final constraint store, which may contain constraint of the form variable=value but in general may contain arbitrary constraints.

Domain-specific constraints may come to the constraint store both from the body of a clauses and from equating a literal with a clause head: for example, if the interpreter rewrites the literal `A(X+2)` with a clause whose fresh variant head is `A(Y/2)`, the constraint `X+2=Y/2` is added to the constraint store. If a variable appears in a real or finite domain expression, it can only take a value in the reals or the finite domain. Such a variable cannot take a term made of a functor applied to other terms as a value. The

constraint store is unsatisfiable if a variable is bound to take both a value of the specific domain and a functor applied to terms.

After a constraint is added to the constraint store, some operations are performed on the constraint store. Which operations are performed depends on the considered domain and constraints. For example unification is used for finite tree equalities, variable elimination for polynomial equations over reals, constraint propagation to enforce a form of local consistency for finite domains. These operations are aimed at making the constraint store simpler to be checked for satisfiability and solved.

As a result of these operations, the addition of new constraints may change the old ones. It is essential that the interpreter is able to undo these changes when it backtracks. The simplest case method is for the interpreter to save the complete state of the store every time it makes a choice (it chooses a clause to rewrite a goal). More efficient methods for allowing the constraint store to return to a previous state exist. In particular, one may just save the changes to the constraint store made between two points of choice, including the changes made to the old constraints. This can be done by simply saving the old value of the constraints that have been modified; this method is called *trailing*. A more advanced method is to save the changes that have been done on the modified constraints. For example, a linear constraint is changed by modifying its coefficient: saving the difference between the old and new coefficient allows reverting a change. This second method is called *semantic backtracking*, because the semantics of the change is saved rather than the old version of the constraints only.

Labeling

The labeling literals are used on variables over finite domains to check satisfiability or partial satisfiability of the constraint store and to find a satisfying assignment. A labeling literal is of the form `labeling([variables])`, where the argument is a list of variables over finite domains. Whenever the interpreter evaluates such a literal, it performs a search over the domains of the variables of the list to find an assignment that satisfies all relevant constraints. Typically, this is done by a form of backtracking: variables are evaluated in order, trying all possible values for each of them, and backtracking when inconsistency is detected.

The first use of the labeling literal is to actual check satisfiability or partial satisfiability of the constraint store. When the interpreter adds a constraint to the constraint store, it only enforces a form of local consistency on it. This operation may not detect inconsistency even if the constraint store is unsatisfiable. A labeling literal over a set of variables enforces a satisfiability check of the constraints over these variables. As a result, using all variables mentioned in the constraint store results in checking satisfiability of the store.

The second use of the labeling literal is to actually determine an evaluation of the variables that satisfies the constraint store. Without the labeling literal, variables are assigned values only when the constraint store contains a constraint of the form `X=value`

and when local consistency reduces the domain of a variable to a single value. A labeling literal over some variables forces these variables to be evaluated. In other words, after the labeling literal has been considered, all variables are assigned a value.

Typically, constraint logic programs are written in such a way labeling literals are evaluated only after as much constraints as possible have been accumulated in the constraint store. This is because labeling literals enforce search, and search is more efficient if there are more constraints to be satisfied. A constraint satisfaction problem is typical solved by a constraint logic program having the following structure:

```
solve(X):-constraints(X), labeling(X)
constraints(X):- (all constraints of the CSP)
```

When the interpreter evaluates the goal `solve(args)`, it places the body of a fresh variant of the first clause in the current goal. Since the first goal is `constraints(X')`, the second clause is evaluated, and this operation moves all constraints in the current goal and eventually in the constraint store. The literal `labeling(X')` is then evaluated, forcing a search for a solution of the constraint store. Since the constraint store contains exactly the constraints of the original constraint satisfaction problem, this operation searches for a solution of the original problem.

Program reformulations

A given constraint logic program may be reformulated to improve its efficiency. A first rule is that labeling literals should be placed after as much constraints on the labeled literals are accumulated in the constraint store. While in theory `A(X):-labeling(X),X>0` is equivalent to `A(X):-X>0,labeling(X)`, the search that is performed when the interpreter encounters the labeling literal is on a constraint store that does not contain the constraint `X>0`. As a result, it may generate solutions, such as `X=-1`, that are later found out not to satisfy this constraint. On the other hand, in the second formulation the search is performed only when the constraint is already in the constraint store. As a result, search only returns solutions that are consistent with it, taking advantage of the fact that additional constraints reduce the search space.

A second reformulation that can increase efficiency is to place constraints before literals in the body of clauses. Again, `A(X):-B(X),X>0` and `A(X):-X>0,B(X)` are in principle equivalent. However, the first may require more computation. For example, if the constraint store contains the constraint `X<-2`, the interpreter recursively evaluates `B(X)` in the first case; if it succeeds, it then finds out that the constraint store is inconsistent when adding `X>0`. In the second case, when evaluating that clause, the interpreter first adds `X>0` to the constraint store and then possibly evaluates `B(X)`. Since the constraint store after the addition of `X>0` turns out to be inconsistent, the recursive evaluation of `B(X)` is not performed at all.

A third reformulation that can increase efficiency is the addition of redundant constrains. If the programmer knows (by whatever means) that the solution of a problem satisfies a

specific constraint, they can include that constraint to cause inconsistency of the constraint store as soon as possible. For example, if it is known beforehands that the evaluation of `B(X)` will result in a positive value for `X`, the programmer may add `X>0` before any occurrence of `B(X)`. As an example, `A(X,Y):-B(X),C(X)` will fail on the goal `A(-2,Z)`, but this is only found out during the evaluation of the subgoal `B(X)`. On the other hand, if the above clause is replaced by `A(X,Y):-X>0,A(X),B(X)`, the interpreter backtracks as soon as the constraint `X>0` is added to the constraint store, which happens before the evaluation of `B(X)` even starts.

Constraint handling rules

Constraint handling rules were initially defined as a stand-alone formalism for specifying constraint solvers, and were later embedded in logic programming. There are two kinds of constraint handling rules. The rules of the first kind specify that, under a given condition, a set of constraints is equivalent to another one. The rules of the second kind specify that, under a given condition, a set of constraints implies another one. In a constraint logic programming language supporting constraint handling rules, a programmer can use these rules to specify possible rewritings of the constraint store and possible additions of constraints to it. The following are example rules:

```
A(X) <=> B(X) | C(X)
A(X) ==> B(X) | C(X)
```

The first rule tells that, if `B(X)` is entailed by the store, the constraint `A(X)` can be rewritten as `C(X)`. As an example, `N*X>0` can be rewritten as `X>0` if the store implies that `N>0`. The symbol `<=>` resembles equivalence in logic, and tells that the first constraint is equivalent to the latter. In practice, this implies that the first constraint can be *replaced* with the latter.

The second rule instead specifies that the latter constraint is a consequence of the first, if the constraint in the middle is entailed by the constraint store. As a result, if `A(X)` is in the constraint store and `B(X)` is entailed by the constraint store, then `C(X)` can be added to the store. Differently from the case of equivalence, this is an addition and not a replacement: the new constraint is added but the old one remains.

Equivalence allows for simplifying the constraint store by replacing some constraints with simpler ones; in particular, if the third constraint in an equivalence rule is `true`, and the second constraint is entailed, the first constraint is removed from the constraint store. Inference allows for the addition of new constraints, which may lead to proving inconsistency of the constraint store, and may generally reduce the amount of search needed to establish its satisfiability.

Logic programming clauses in conjunction with constraint handling rules can be used to specify a method for establishing the satisfiability of the constraint store. Different clauses are used to implement the different choices of the method; the constraint handling rules are used for rewriting the constraint store during execution. As an example, one can

implement backtracking with unit propagation this way. Let `holds(L)` represents a propositional clause, in which the literals in the list `L` are in the same order as they are evaluated. The algorithm can be implemented using clauses for the choice of assigning a literal to true or false, and constraint handling rules to specify propagation. These rules specify that `holds([l|L])` can be removed if `l=true` follows from the store, and it can be rewritten as `holds(L)` if `l=false` follows from the store. Similarly, `holds([l])` can be replaced by `l=true`. In this example, the choice of value for a variable is implemented using clauses of logic programming; however, it can be encoded in constraint handling rules using an extension called disjunctive constraint handling rules or CHR$^{\vee}$.

Bottom-up evaluation

The standard strategy of evaluation of logic programs is top-down and depth-first: from the goal, a number of clauses are identified as being possibly able to prove the goal, and recursion over the literals of their bodies is performed. An alternative strategy is to start from the facts and use clauses to derive new facts; this strategy is called bottom-up. It is considered better than the top-down one when the aim is that of producing all consequences of a given program, rather than proving a single goal. In particular, finding all consequences of a program in the standard top-down and depth-first manner may not terminate while the bottom-up evaluation strategy terminates.

The bottom-up evaluation strategy maintains the set of facts proved so far during evaluation. This set is initially empty. With each step, new facts are derived by applying a program clause to the existing facts, and are added to the set. For example, the bottom up evaluation of the following program requires two steps:

```
A(q).
B(X):-A(X).
```

The set of consequences is initially empty. At the first step, `A(q)` is the only clause whose body can be proved (because it is empty), and `A(q)` is therefore added to the current set of consequences. At the second step, since `A(q)` is proved, the second clause can be used and `B(q)` is added to the consequences. Since no other consequence can be proved from `{A(q),B(q)}`, execution terminates.

The advantage of the bottom-up evaluation over the top-down one is that cycles of derivations do not produce an infinite loop. This is because adding a consequence to the current set of consequences that already contains it has no effect. As an example, adding a third clause to the above program generates a cycle of derivations in the top-down evaluation:

```
A(q).
B(X):-A(X).
A(X):-B(X).
```

For example, while evaluating all answers to the goal `A(X)`, the top-down strategy would produce the following derivations:

```
A(q)
A(q):-B(q), B(q):-A(q), A(q)
A(q):-B(q), B(q):-A(q), A(q):-B(q), B(q):-A(q), A(q)
```

In other words, the only consequence `A(q)` is produced first, but then the algorithm cycles over derivations that do not produce any other answer. More generally, the top-down evaluation strategy may cycle over possible derivations, possibly when other ones exist.

The bottom-up strategy does not have the same drawback, as consequences that were already derived has no effect. On the above program, the bottom-up strategy starts adding `A(q)` to the set of consequences; in the second step, `B(X):-A(X)` is used to derive `B(q)`; in the third step, the only facts that can be derived from the current consequences are `A(q)` and `B(q)`, which are however already in the set of consequences. As a result, the algorithm stops.

In the above example, the only used facts were ground literals. In general, every clause that only contains constraints in the body is considered a fact. For example, a clause `A(X):-X>0,X<10` is considered a fact as well. For this extended definition of facts, some facts may be equivalent while not syntactically equal. For example, `A(q)` is equivalent to `A(X):-X=q` and both are equivalent to `A(X):-X=Y, Y=q`. To solve this problem, facts are translated into a normal form in which the head contains a tuple of all-different variables; two facts are then equivalent if their bodies are equivalent on the variables of the head, that is, their sets of solutions are the same when restricted to these variables.

As described, the bottom-up approach has the advantage of not considering consequences that have already been derived. However, it still may derive consequences that are entailed by those already derived while not being equal to any of them. As an example, the bottom up evaluation of the following program is infinite:

```
A(0).
A(X):-X>0.
A(X):-X=Y+1, A(Y).
```

The bottom-up evaluation algorithm first derives that `A(X)` is true for `X=0` and `X>0`. In the second step, the first fact with the third clause allows for the derivation of `A(1)`. In the third step, `A(2)` is derived, etc. However, these facts are already entailed by the fact that `A(X)` is true for any nonnegative `X`. This drawback can be overcome by checking for entailment facts that are to be added to the current set of consequences. If the new consequence is already entailed by the set, it is not added to it. Since facts are stored as clauses, possibly with "local variables", entailment is restricted over the variables of their heads.

Concurrent constraint logic programming

Concurrent constraint logic programming is a version of constraint logic programming aimed primarily at programming concurrent processes rather than (or in addition to) solving constraint satisfaction problems. Goals in constraint logic programming are evaluated concurrently; a concurrent process is therefore programmed as the evaluation of a goal by the interpreter.

Syntactically, concurrent constraints logic programs are similar to non-concurrent programs, the only exception being that clauses include guards, which are constraints that may block the applicability of the clause under some conditions. Semantically, concurrent constraint logic programming differs from its non-concurrent versions because a goal evaluation is intended to realize a concurrent process rather than finding a solution to a problem. Most notably, this difference affects how the interpreter behaves when more than one clause is applicable: non-concurrent constraint logic programming recursively tries all clauses; concurrent constraint logic programming chooses only one. This is the most evident effect of an intended *directionality* of the interpreter, which never revise a choice it has previously taken. Other effects of this are the semantical possibility of having a goal that cannot be proved while the whole evaluation does not fail, and a particular way for equating a goal and a clause head.

Constraint handling rules can be seen as a form of concurrent constraint logic programming, but are used for programming a constraint simplifier or solver rather than concurrent processes.

Description

In constraint logic programming, the goals in the current goal are evaluated sequentially, usually proceeding in a LIFO order in which newer goals are evaluated first. The concurrent version of logic programming allows for evaluating goals in parallel: every goal is evaluated by a process, and processes run concurrently. These processes interact via the constraint store: a process can add a constraint to the constraint store while another one checks whether a constraint is entailed by the store.

Adding a constraint to the store is done like in regular constraint logic programming. Checking entailment of a constraint is done via guards to clauses. Guards require a syntactic extension: a clause of concurrent constraint logic programming is written as `H :- G | B` where `G` is a constraint called the guard of the clause. Roughly speaking, a fresh variant of this clause can be used to replace a literal in the goal only if the guard is entailed by the constraint store after the equation of the literal and the clause head is added to it. The precise definition of this rule is more complicated, and is given below.

The main difference between non-concurrent and concurrent constraint logic programming is that the first is aimed at search, while the second is aimed at implementing concurrent processes. This difference affects whether choices can be

undone, whether processes are allowed not to terminate, and how goals and clause heads and equated.

The first semantical difference between regular and concurrent constraint logic programming is about the condition when more than one clause can be used for proving a goal. Non-concurrent logic programming tries all possible clauses when rewriting a goal: if the goal cannot be proved while replacing it with the body of a fresh variant of a clause, another clause is proved, if any. This is because the aim is to prove the goal: all possible ways to prove the goal are tried. On the other hand, concurrent constraint logic programming aims at programming parallel processes. In general concurrent programming, if a process makes a choice, this choice cannot be undone. The concurrent version of constraint logic programming implements processes by allowing them to take choices, but committing to them once they have been taken. Technically, if more than one clause can be used to rewrite a literal in the goal, the non-concurrent version tries in turn all clauses, while the concurrent version chooses a single arbitrary clause: contrary to the non-concurrent version, the other clauses will never be tried. These two different ways for handling multiple choices are often called "don't know nondeterminism" and "don't care nondeterminism".

When rewriting a literal in the goal, the only considered clauses are those whose guard is entailed by the union of the constraint store and the equation of the literal with the clause head. The guards provide a way for telling which clauses are not be considered at all. This is particularly important given the commitment to a single clause of concurrent constraint logic programming: once a clause has been chosen, this choice will be never reconsidered. Without guards, the interpreter could choose a "wrong" clause to rewrite a literal, while other "good" clauses exist. In non-concurrent programming, this is less important, as the interpreter always tries all possibilities. In concurrent programming, the interpreter commits to a single possibility without trying the other ones.

A second effect of the difference between the non-concurrent and the concurrent version is that concurrent constraint logic programming is specifically designed to allow processes to run without terminating. Non-terminating processes are common in general in concurrent processing; the concurrent version of constraint logic programming implements them by not using the condition of failure: if no clause is applicable for rewriting a goal, the process evaluating this goal stops instead of making the whole evaluation fail like in non-concurrent constraint logic programming. As a result, the process evaluating a goal may be stopped because no clause is available to proceed, but at the same time the other processes keep running.

Synchronization among processes that are solving different goals is achieved via the use of guards. If a goal cannot be rewritten because all clauses that could be used have a guard that is not entailed by the constraint store, the process solving this goal is blocked until the other processes add the constraints that are necessary to entail the guard of at least one of the applicable clauses. This synchronization is subject to deadlocks: if all goals are blocked, no new constraints will be added and therefore no goal will ever be unblocked.

A third effect of the difference between concurrent and non-concurrent logic programming is in the way a goal is equated to the head of a fresh variant of a clause. Operationally, this is done by checking whether the variables in the head can be equated to terms in such a way the head is equal to the goal. This rule differs from the corresponding rule for constraint logic programming in that it only allows adding constraints in the form variable=term, where the variable is one of the head. This limitation can be seen as a form of directionality, in that the goal and the clause head are treated differently.

Precisely, the rule telling whether a fresh variant `H:-G|B` of a clause can be used to rewrite a goal `A` is as follows. First, it is checked whether `A` and `H` have the same predicate. Second, it is checked whether there exists a way for equating *A* with *H* given the current constraint store; contrary to regular logic programming, this is done under *one-sided unification*, which only allows a variable of the head to be equal to a term. Third, the guard is checked for entailment from the constraint store and the equations generated in the second step; the guard may contain variables that are not mentioned in the clause head: these variables are interpreted existentially. This method for deciding the applicability of a fresh variant of a clause for replacing a goal can be compactly expressed as follows: the current constraint store entails that there exists an evaluation of the variables of the head and the guard such that the head is equal to the goal and the guard is entailed. In practice, entailment may be checked with an incomplete method.

An extension to the syntax and semantics of concurrent logic programming is the *atomic tell*. When the interpreter uses a clause, its guard is added to the constraint store. However, also added are the constraints of the body. Due to commitment to this clause, the interpreter does not backtrack if the constraints of the body are inconsistent with the store. This condition can be avoided by the use of atomic tell, which is a variant in which the clause contain a sort of "second guard" that is only checked for consistency. Such a clause is written `H :- G:D|B`. This clause is used to rewrite a literal only if `G` is entailed by the constraint store and `D` is consistent with it. In this case, both `G` and `D` are added to the constraint store.

History

The study of concurrent constraint logic programming started at the end of the 1980s, when some of the principles of concurrent logic programming were integrated into constraint logic programming by Michael J. Maher. The theoretical properties of concurrent constraint logic programming were later studied by various authors, such as Vijay A. Saraswat.

Applications

Constraint logic programming has been applied to a number of fields, such as civil engineering, mechanical engineering, digital circuit verification, automated timetabling, air traffic control, finance, and others.

Chapter 6

Linear Logic

Linear logic is a substructural logic proposed by Jean-Yves Girard as a refinement of classical and intuitionistic logic, joining the dualities of the former with many of the constructive properties of the latter. Although the logic has also been studied for its own sake, more broadly, ideas from linear logic have been influential in fields such as programming languages, game semantics, and quantum physics, particularly because of its emphasis on resource-boundedness, duality, and interaction.

Linear logic lends itself to many different presentations, explanations and intuitions. Proof-theoretically, it derives from an analysis of classical sequent calculus in the absence of the structural rules of weakening and contraction. (This has the effect that certain propositions which are classically/intuitionistically valid are not directly provable in linear logic, although both classical and intuitionistic logic can be encoded in linear logic by means of additional modal connectives, the so-called *exponentials*.) Operationally, the rejection of weakening and contraction can be seen as reorienting the subject of logic, from persistent *truths* to ephemeral *resources*. Logical deduction then corresponds to local (possibly destructive) transformations on these resources, rather than the usual view of deduction as building up an ever-expanding collection of facts. Denotationally, linear logic can be seen as refining the interpretation of intuitionistic logic by replacing cartesian closed categories by symmetric monoidal categories, or the interpretation of classical logic by replacing boolean algebras by C*-algebras.

Connectives, duality, and polarity

Syntax

The language of *classical linear logic* (CLL) is defined inductively by the BNF notation

$$
\begin{aligned}
A \ ::= \ & p \mid p^{\perp} \\
\mid \ & A \otimes A \mid A \oplus A \\
\mid \ & A \,\&\, A \mid A \ \ A \\
\mid \ & 1 \mid 0 \mid \top \mid \bot \\
\mid \ & !A \mid ?A
\end{aligned}
$$

Here p and $p^{\perp}$ range over logical atoms. For reasons to be explained below, the connectives $\otimes$, 1, and $\perp$ are called *multiplicatives*, the connectives &, $\oplus$, $\top$, and 0 are called *additives*, and the connectives ! and ? are called *exponentials*. We can further employ the following terminology:

- $\otimes$ is called "multiplicative conjunction" or "times" (or sometimes "tensor")
- $\oplus$ is called "additive disjunction" or "plus"
- & is called "additive conjunction" or "with"
- ! is pronounced "of course" (or sometimes "bang")
- ? is pronounced "why not"

Every proposition A in CLL has a **dual** $A^{\perp}$, defined as follows:

$(p)^{\perp} = p^{\perp}$	$(p^{\perp})^{\perp} = p$
$(A \otimes B)^{\perp} = A^{\perp} \& B^{\perp}$	$(A \& B)^{\perp} = A^{\perp} \otimes B^{\perp}$
$(A \oplus B)^{\perp} = A^{\perp} \& B^{\perp}$	$(A \& B)^{\perp} = A^{\perp} \oplus B^{\perp}$
$(1)^{\perp} = \perp$	$(\perp)^{\perp} = 1$
$(0)^{\perp} = \top$	$(\top)^{\perp} = 0$
$(!A)^{\perp} = ?A^{\perp}$	$(?A)^{\perp} = !A^{\perp}$

Observe that $(-)^{\perp}$ is an involution, i.e., $A^{\perp\perp} = A$ for all propositions. $A^{\perp}$ is also called the *linear negation* of A.

The columns of the table suggest another way of classifying the connectives of linear logic, termed **polarity**: the connectives negated in the left column ($\otimes$, $\oplus$, 1, 0, !) are called *positive*, while their duals on the right (&, $\perp$, $\top$, ?) are called *negative*.

Linear implication is not included in the grammar of connectives, but is definable in CLL using linear negation and multiplicative disjunction, by $A \multimap B := A^{\perp} \& B$. The connective $\multimap$ is sometimes pronounced "lollipop", owing to its shape.

Sequent calculus presentation

One way of fixing the formal meaning of the connectives is by explaining how to prove linear logic propositions. Here we follow Girard's original presentation of CLL as a one-sided sequent calculus (in the style of Schütte and Tait).

A *context* (Γ, Δ) is a list of propositions A_1, ..., A_n. A (one-sided) *sequent* is the assertion $\vdash \Gamma$ of a context. We give inference rules describing how to build proofs of sequents.

First, to formalize the fact that we do not care about the order of propositions inside a context, we add the structural rule of exchange:

$$\frac{\vdash \Gamma}{\vdash \Gamma'} \ (\Gamma' \text{ a permutation of } \Gamma)$$

(Alternatively we could accomplish the same thing by declaring contexts to be multisets rather than lists.) Note that we do **not** add the structural rules of weakening and contraction, because we do care about the absence of propositions in a sequent, and the number of copies present.

Next we add *initial sequents* and *cuts*:

$$\frac{}{\vdash A, A^{\perp}} \qquad \frac{\vdash \Gamma, A \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}$$

The cut rule can be seen as a way of composing proofs, and initial sequents serve as the units for composition. In a certain sense these rules are redundant: as we introduce additional rules for building proofs below, we will maintain the property that arbitrary initial sequents can be derived from atomic initial sequents, and that whenever a sequent is provable it can be given a cut-free proof. Ultimately, this canonical form property (which can be divided into the completeness of atomic initial sequents and the cut-elimination theorem, inducing a notion of analytic proof) lies behind the applications of linear logic in computer science, since it allows the logic to be used in proof search and as a resource-aware lambda-calculus.

Now, we explain the connectives by giving *logical rules*. Typically in sequent calculus one gives both "right-rules" and "left-rules" for each connective, essentially describing two modes of reasoning about propositions involving that connective (e.g., verification and falsification). In a one-sided presentation, one instead makes use of negation: the right-rules for a connective effectively play the role of left-rules for its dual ($\otimes$). So, we should expect a certain "harmony" between the rule(s) for a connective and the rule(s) for its dual.

Multiplicatives

The rules for multiplicative conjunction ($\otimes$) and disjunction ($\&$):

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{} \qquad \frac{\vdash \Gamma, A, B}{}$$

$$\vdash \Gamma, \Delta, A \otimes B \qquad\qquad \vdash \Gamma, A \parr B$$

and for their units:

$$\frac{}{\vdash 1} \qquad\qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}$$

Observe that the rules for multiplicative conjunction and disjunction are admissible for plain *conjunction* and *disjunction* under a classical interpretation (i.e., they are admissible rules in LK).

Additives

The rules for additive conjunction (&) and disjunction (⊕) :

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B} \qquad\qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}$$

and for their units:

$$\frac{}{\vdash \Gamma, \top} \qquad \text{(no rule for 0)}$$

Observe that the rules for additive conjunction and disjunction are again admissible under a classical interpretation. But now we can explain the basis for the multiplicative/additive distinction in the rules for the two different versions of conjunction: for the multiplicative connective (⊗), the context of the conclusion (Γ, Δ) is split up between the premises, whereas for the additive case connective (&) the context of the conclusion (Γ) is carried whole into both premises.

Exponentials

The exponentials are used to give controlled access to weakening and contraction. Specifically, we add structural rules of weakening and contraction for ?'d propositions:

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?A} \qquad\qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}$$

$$\vdash \Gamma, ?A \qquad \vdash \Gamma, ?A$$

and use the following logical rules:

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}$$

One might observe that the rules for the exponentials follow a different pattern from the rules for the other connectives, and that there is no longer such a clear symmetry between the duals ! and ?. This situation is remedied in alternative presentations of CLL (e.g., the LU presentation).

Remarkable formulae

In addition to the De Morgan dualities described above, some important equivalences in linear logic include:

Distributivity

$$A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C)$$

Exponential isomorphism

$$!(A \,\&\, B) \equiv !A \otimes !B$$

(Here $A \equiv B \quad = \quad (A \multimap B) \& (B \multimap A)$.)

The following is not in general an equivalence, only an implication:

Semi-distributivity

$$(A \otimes (B \parr C)) \multimap ((A \otimes B) \parr C)$$

Encoding classical/intuitionistic logic in linear logic

Both intuitionistic and classical implication can be recovered from linear implication by inserting exponentials: intuitionistic implication is encoded as $!A \multimap B$, and classical implication as $!A \multimap ?B$.

The resource interpretation

Lafont (1993) first showed how intuitionistic linear logic can be explained as a logic of resources, so providing the logical language with access to formalisms that can be used for reasoning about resources within the logic itself, rather than, as in classical logic, by means of non-logical predicates and relations. Sir Antony Hoare (1985)'s classical example of the vending machine can be used to illustrate this idea.

Suppose we represent a candy bar by the atomic proposition *candy*, and a dollar by *$1*. To state the fact that a dollar will buy you one candy bar, we might write the implication $\$1 \Rightarrow candy$. But in ordinary (classical or intuitionistic) logic, from A and $A \Rightarrow B$ one can conclude $A \wedge B$. So, ordinary logic leads us to believe that we can buy the candy bar and keep our dollar! Of course, we can avoid this problem by using more sophisticated encodings, although typically such encodings suffer from the frame problem. However, the rejection of weakening and contraction allows linear logic to avoid this kind of spurious reasoning even with the "naive" rule. Rather than $\$1 \Rightarrow candy$, we express the property of the vending machine as a *linear* implication $\$1 \multimap candy$. From *$1* and this fact, we can conclude *candy*, but *not* $\$1 \otimes candy$. In general, we can use the linear logic proposition $A \multimap B$ to express the validity of transforming resource A into resource B.

Running with the example of the vending machine, let us consider the "resource interpretations" of the other multiplicative and additive connectives. (The exponentials provide the means to combine this resource interpretation with the usual notion of persistent logical truth.)

Multiplicative conjunction ($A \otimes B$) denotes simultaneous occurrence of resources, to be used as the consumer directs. For example, if you buy a stick of gum and a bottle of soft drink, then you are requesting $gum \otimes drink$. The constant 1 denotes the absence of any resource, and so functions as the unit of $\otimes$.

Additive conjunction ($A \,\&\, B$) represents alternative occurrence of resources, the choice of which the consumer controls. If in the vending machine there is a packet of chips, a candy bar, and a can of soft drink, each costing one dollar, then for that price you can buy exactly one of these products. Thus we write $\$1 \multimap (candy \,\&\, chips \,\&\, drink)$. We do *not* write $\$1 \multimap (candy \otimes chips \otimes drink)$, which would imply that one dollar suffices for buying all three products together. However, from $\$1 \multimap (candy \,\&\, chips \,\&\, drink)$, we can correctly deduce $\$3 \multimap (candy \otimes chips \otimes drink)$, where $\$3 := \$1 \otimes \$1 \otimes \$1$. The unit $\top$ of additive conjunction can be seen as a wastebasket or garbage collector for irrelevant alternatives. For example, we can write $\$3 \multimap (candy \otimes \top)$ to express that three dollars will buy you a candy bar and something else (we don't care what).

Additive disjunction ($A \oplus B$) represents alternative occurrence of resources, the choice of which the machine controls. For example, suppose the vending machine permits gambling: insert a dollar and the machine may dispense a candy bar, a packet of chips, or a soft drink. We can express this situation as $\$1 \multimap (candy \oplus chips \oplus drink)$. The constant 0 represents a product that cannot be made, and thus serves as the unit of $\oplus$ (a machine that might produce A or 0 is as good as a machine that always produces A because it will never succeed in producing a 0).

Multiplicative disjunction ($A \,\&\, B$) is more difficult to gloss in terms of the resource interpretation, although we can encode back into linear implication, either as $A^{\perp} \multimap B$ or $B^{\perp} \multimap A$.

Decidability/complexity of entailment

The entailment relation in full CLL is undecidable. Fragments of CLL are often considered, for which the decision problem is more subtle:

- Multiplicative linear logic (MLL): only the multiplicative connectives. MLL entailment is NP-complete.
- Multiplicative-additive linear logic (MALL): only multiplicatives and additives (i.e., exponential-free). MALL entailment is PSPACE-complete.
- Multiplicative-exponential linear logic (MELL): only multiplicatives and exponentials. The decidability of MELL entailment is currently open.

Variants of linear logic

Many variations of linear logic arise by further tinkering with the structural rules:

- Affine logic, which forbids contraction but allows global weakening.
- Strict logic or relevant logic, which forbids weakening but allows global contraction.
- Non-commutative logic or ordered logic, which removes the rule of exchange, in addition to barring weakening and contraction. In ordered logic, linear implication divides further into left-implication and right-implication.

Different intuitionistic variants of linear logic have been considered. When based on a single-conclusion sequent calculus presentation, like in ILL (Intuitionistic Linear Logic), the connectives &, ⊥, and ? are absent, and linear implication is treated as a primitive connective. In FILL (Full Intuitionistic Linear Logic) the connectives &, ⊥, and ? are present, linear implication is a primitive connective and, similarly to what happens in intuitionistic logic, all connectives (except linear negation) are independent. There are also first- and higher-order extensions of linear logic, whose formal development is somewhat standard.

Chapter 7

Introduction to Type Theory

In mathematics, logic and computer science, **type theory** is any of several formal systems that can serve as alternatives to naive set theory, or the study of such formalisms in general. In programming language theory, a branch of computer science, *type theory* can refer to the design, analysis and study of type systems, although some computer scientists limit the term's meaning to the study of abstract formalisms such as typed λ-calculi.

Bertrand Russell invented the first type theory in response to his discovery that Gottlob Frege's version of naive set theory was afflicted with Russell's paradox. This theory of types features prominently in Whitehead and Russell's *Principia Mathematica*. It avoids Russell's paradox by first creating a hierarchy of types, then assigning each mathematical (and possibly other) entity to a type. Objects of a given type are built exclusively from objects of preceding types (those lower in the hierarchy), thus preventing loops.

Alonzo Church, inventor of the lambda calculus, developed a higher-order logic commonly called *Church's Theory of Types*, in order to avoid the Kleene–Rosser paradox afflicting the original pure lambda calculus. Church's type theory is a variant of the lambda calculus in which expressions (also called formulas or λ-terms) are classified into types, and the types of expressions restrict the ways in which they can be combined. In other words, it is a typed lambda calculus. Today many other such calculi are in use, including Per Martin-Löf's Intuitionistic type theory, Jean-Yves Girard's System F and the Calculus of Constructions. In typed lambda calculi, types play a role similar to that of sets in set theory.

Simple theory of types

In the 1920s, Leon Chwistek and Frank P. Ramsey noticed that, if one is willing to give up the vicious circle principle, the hierarchy of levels of types in the "ramified theory of types" can be collapsed. The resulting simplified logic is called the simple theory of types or, more briefly, simple type theory. ST is equivalent with Russell's ramified theory plus the Axiom of reducibility. Detailed formulations of simple type theory were published in the late 1920s and early 1930s by R. Carnap, K. Gödel, W.V.O. Quine, and A. Tarski. In 1940 Alonzo Church (re)formulated it as simply typed lambda calculus.

The following system is Mendelson's (1997, 289–293) **ST**. The domain of quantification is partitioned into an ascending hierarchy of types, with all individuals assigned a type. Quantified variables range over only one type; hence the underlying logic is first-order logic. **ST** is "simple" (relative to the type theory of *Principia Mathematica*) primarily because all members of the domain and codomain of any relation must be of the same type. There is a lowest type, whose individuals have no members and are members of the second lowest type. Individuals of the lowest type correspond to the urelements of certain set theories. Each type has a next higher type, analogous to the notion of successor in Peano arithmetic. While **ST** is silent as to whether there is a maximal type, a transfinite number of types poses no difficulty. These facts, reminiscent of the Peano axioms, make it convenient and conventional to assign a natural number to each type, starting with 0 for the lowest type. But type theory does not require a prior definition of the naturals.

The symbols peculiar to **ST** are primed variables and infix $\in$. In any given formula, unprimed variables all have the same type, while primed variables (*x*') range over the next higher type. The atomic formulas of **ST** are of two forms, $x = y$ (identity) and $y \in x'$. The infix symbol $\in$suggests the intended interpretation, set membership.

All variables appearing in the definition of identity and in the axioms *Extensionality* and *Comprehension*, range over individuals of one of two consecutive types. Only unprimed variables (ranging over the "lower" type) can appear to the left of '$\in$', where as to its right, only primed variables (ranging over the "higher" type) can appear. The first-order formulation of **ST** rules out quantifying over types. Hence each pair of consecutive types requires its own axiom of Extensionality and of Comprehension, which is possible if *Extensionality* and *Comprehension* below are taken as axiom schemata "ranging over" types.

- **Identity**, defined by $x = y \leftrightarrow \forall z'[x \in z' \leftrightarrow y \in z']$.
- **Extensionality**. An axiom schema. $\forall x[x \in y' \leftrightarrow x \in z'] \rightarrow y' = z'$.

  Let $\Phi(x)$ denote any first-order formula containing the free variable *x*.

- **Comprehension**. An axiom schema. $\exists z' \forall x[x \in z' \leftrightarrow \Phi(x)]$.

  Remark. Any collection of elements of the same type may form an object of the next higher type. Comprehension is schematic with respect to $\Phi(x)$ as well as to types.

- **Infinity**. There exists a nonempty binary relation *R* over the individuals of the lowest type, that is irreflexive, transitive, and strongly connected: $\forall x, y[x \neq y \rightarrow [xRy \vee yRx]]$.

  Remark. Infinity is the only true axiom of **ST** and is entirely mathematical in nature. It asserts that *R* is a strict total order, with a domain identical to its

codomain. If 0 is assigned to the lowest type, the type of *R* is 3. Infinity can be satisfied only if the (co)domain of *R* is infinite, thus forcing the existence of an infinite set. If relations are defined in terms of ordered pairs, this axiom requires a prior definition of ordered pair; the Kuratowski definition, adapted to **ST**, will do. The literature does not explain why the usual axiom of infinity (there exists an inductive set) of ZFC of other set theories could not be married to **ST**.

ST reveals how type theory can be made very similar to axiomatic set theory. Moreover, the more elaborate ontology of **ST**, grounded in what is now called the "iterative conception of set," makes for axiom (schemata) that are far simpler than those of conventional set theories, such as ZFC, with simpler ontologies. Set theories whose point of departure is type theory, but whose axioms, ontology, and terminology differ from the above, include New Foundations and Scott–Potter set theory.

Formulations based on equality

Church's type theory has been extensively studied by two of Church's students, Leon Henkin and Peter B. Andrews. Since **ST** is a higher order logic, and in higher order logics one can define propositional connectives in terms of logical equivalence and quantifiers, in 1963 Henkin developed a formulation of **ST** based on equality, but in which he restricted attention to propositional types. This was simplified later that year by Andrews in his theory $\mathbf{Q_0}$. In this respect **ST** can be seen as a particular kind of a higher-order logic, classified by P.T. Johnstone in *Sketches of an Elephant*, as having a lambda-signature, that is a higher-order signature that contains no relations, and uses only products and arrows (function types) as type constructors. Furthermore, as Johnstone put it, **ST** is "logic-free" in the sense that it contains no logical connectives or quantifiers in its formulae.

History

1900 - 1927

Origin of Russell's Theory of Types: In a letter to Gottlob Frege (1902) Russell announced his discovery of the paradox in Frege's Begriffsschrift. Frege promptly responded, acknowledging the problem and proposing a solution in a technical discussion of "levels". To quote Frege: "Incidentally, it seems to me that the expression "a predicate is predicated of itself" is not exact. A predicate is as a rule a first-level function, and this function requires an object as argument and cannot have itself as argument (subject). Therefore I would prefer to say "a concept is predicated of its own extension". He goes about showing how this might work but seems to pull back from it. (In a footnote, van Heijenoort notes that in Frege *1893* Frege had used a symbol (horseshoe) "for reducing second-level functions to first-level functions"). As a consequence of what has become known as Russell's paradox both Frege and Russell had to quickly emend works that they had at the printers. In an Appendix B that Russell tacked on to his 1903 Principles of Mathematics one finds his "tentative" "theory of types".

The matter plagued Russell for about five years (1903–1908). Willard Quine in his preface to Russell's (1908a) *Mathematical logic as based on the theory of types* presents a historical synopsis of the origin of the theory of types and the "ramified" theory of types: Russell proposed in turn a number of alternatives: (i) abandoning the theory of types (1905) followed by three theories in 1905: (ii.1) the zigzag theory, (ii.2) theory of limitation of size, (ii.3) the no-class theory (1905–1906), then (iii) readopting the theory of types (1908ff)".

Quine observes that Russell's introduction of the notion of "apparent variable" had the following result: "the distinction between 'all' and 'any': 'all' is expressed by the bound ('apparent') variable of universal quantification, which ranges over a type, and 'any' is expressed by the free ('real') variable which refers schematically to any unspecified thing irrespective of type". Quine dismisses this notion of "bound variable" as "pointless apart from a certain aspect of the theory of types".

The 1908 "ramified" theory of types

Quine explains the *ramified* theory as follows: "It has been so called because the type of a function depends both on the types of its arguments and on the types of the apparent variables contained in it (or in its expresion), in case these exceed the types of the arguments". Stephen Kleene in his 1952 *Introduction to Metamathematics* describes the *ramified* theory of types this way:

The primary objects or individuals (i.e. the given things not being subjected to logical analysis) are assigned to one type (say *type 0*), the properties of individuals to *type 1*, properties of properties of individuals to *type 2*, etc.; and no properties are admitted which do not fall into one of these logical types (e.g. this puts the properties 'predicable' and 'impredicable' ... outside the pale of logic). A more detailed account would describe the admitted types for other objects as relations and classes. Then to exclude impredicative definitions within a type, the types above type 0 are further separated into orders. Thus for type 1, properties defined without mentioning any totality belong to *order 0*, and properties defined using the totality of properties of a given order belong to the next higher order. ... But this separation into orders makes it impossible to construct the familiar analysis, which we saw above contains impredicative definitions. To escape this outcome, Russell postulated his *axiom of reducibility*, which asserts that to any property belonging to an order above the lowest, there is a coextensive property (i.e. one possessed by exactly the same objects) of order 0. If only definable properties are considered to exist, then the axiom means that to every impredicative definition within a given type there is an equivalent predicative one (Kleene 1952:44-45).

The axiom of reducibility and the notion of "matrix"

But because the stipulations of the ramified theory would prove (to quote Quine) "onerous", Russell in his 1908 *Mathematical logic as based on the theory of types* also would propose his *axiom of reducibility*. By 1910 Whitehead and Russell in their *Principia Mathematica* would further augment this axiom with the notion of a *matrix* -- a

fully-extensional specification of a function. From its matrix a function could be derived by the process of "generalization" and vice versa, i.e. the two processes are reversible -- (i) generalization from a matrix to a function (by use apparent variables) and (ii) the reverse process of reduction of type by courses-of-values substitution of arguments for the apparent variable. By this method impredicativity could be avoided.

Truth tables

Eventually Emil Post (1921) would lay waste to Russell's "cumbersome" Theory of Types with his "truth functions" and their truth tables. In his "Introduction" to his 1921 Post places the blame on Russell's notion of apparent variable: "Whereas the complete theory [of Whitehead and Russell (1910, 1912, 1913)] requires for the enunciation of its propositions real and apparent variables, which represent both individuals and propositional functions of different kinds, and as a result necessitates the cumbersome theory of types, this subtheory uses only real variables, and these real variables represent but one kind of entity, which the authors have chosen to call elementary propositions".

At about the same time Ludwig Wittgenstein made short work of the theory of types in his 1922 work Tractatus Logico-Philosophicus in which he points out the following in parts 3.331–3.333:

3.331 From this observation we get a further view – into Russell's Theory of Types. Russell's error is shown by the fact that in drawing up his symbolic rules he has to speak of the meanings of his signs.

3.332 No proposition can say anything about itself, because the propositional sign cannot be contained in itself (that is the whole "theory of types").

3.333 A function cannot be its own argument, because the functional sign already contains the prototype of its own argument and it cannot contain itself...

Wittgenstein proposed the truth-table method as well. In his 4.3 through 5.101, Wittgenstein adopts an unbounded Sheffer stroke as his fundamental logical entity and then lists all 16 functions of two variables (5.101).

Russell's doubts

Russell in his 1920 *Introduction to Mathematical Philosophy* devotes an entire chapter to "The axiom of Infinity and logical types" wherein he states his concerns: "Now the theory of types emphatically does not belong to the finished and certain part of our subject: much of this theory is still inchoate, confused, and obscure. But the need of *some* doctrine of types is less doubtful than the precise form the doctrine should take; and in connection with the axiom of infinity it is particularly easy to see the necessity of some such doctrine".

Russell abandons the axiom of reducibility: In the second edition of *Principia Mathematica* (1927) he acknowledges Wittgenstein's argument. At the outset of his Introduction he declares "there can be no doubt ... that there is no need of the distinction between real and apparent variables...". Now he fully embraces the matrix notion and declares "A *function can only appear in a matrix through its values*" (but demurs in a footnote: "It takes the place (not quite adequately) of the axiom of reducibility"). Furthermore, he introduces a new (abbreviated, generalized) notion of "matrix", that of a "logical matrix . . . one that contains no constants. Thus *p*|*q* is a logical matrix". Thus Russell has virtually abandoned the axiom of reducibility, but in his last paragraphs he states that from "our present primitive propositions" he cannot derive "Dedekindian relations and well-ordered relations" and observes that if there is a new axiom to replace the axiom of reducibility "it remains to be discovered".

1927 - present

Practical impact

Computing

The most obvious application of type theory is in constructing type checking algorithms in the semantic analysis phase of compilers for programming languages.

Linguistics

Type theory is also widely in use in theories of semantics of natural language, especially Montague grammar and its descendants. The most common construction takes the basic types *e* and *t* for individuals and truth-values, respectively, and defines the set of types recursively as follows:

- if *a* and *b* are types, then so is $\langle a, b \rangle$.
- Nothing except the basic types, and what can be constructed from them by means of the previous clause are types.

A complex type $\langle a, b \rangle$is the type of functions from entities of type *a* to entities of type *b*. Thus one has types like $\langle e, t \rangle$which are interpreted as elements of the set of functions from entities to truth-values, i.e. characteristic functions of sets of entities. An expression of type $\langle \langle e, t \rangle, t \rangle$is a function from sets of entities to truth-values, i.e. a (characteristic function of a) set of sets. This latter type is standardly taken to be the type of natural language quantifiers, like *everybody* or *nobody* (Montague 1973, Barwise and Cooper 1981).

Social sciences

Gregory Bateson introduced a theory of logical types into the social sciences; his notions of double bind and logical levels are based on Russell's theory of types.

Connections to constructive logic

Relation to other topics

Type system

Definitions of *type system* vary, but the following one due to Benjamin C. Pierce roughly corresponds to the current consensus in the programming language theory community:

[A type system is a] tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

In other words, a type system divides program values into sets called *types* — this is called a *type assignment* — and makes certain program behaviors illegal on the basis of the types that are thus assigned. For example, a type system may classify the value "hello" as a string and the value 5 as a number, and prohibit the programmer from adding "hello" to 5 based on that type assignment. In this type system, the program

```
"hello" + 5
```

would be illegal. Hence, any program permitted by the type system would be provably free from the erroneous behavior of adding strings and numbers.

The design and implementation of type systems is a topic nearly as broad as the topic of programming languages itself. In fact, type theory proponents commonly proclaim that the design of type systems is the very essence of programming language design: "Design the type system correctly, and the language will design itself."

Chapter 8

Simply Typed Lambda Calculus

The **simply typed lambda calculus** ($\lambda^{\rightarrow}$) is a typed interpretation of the lambda calculus with only one type constructor: $\rightarrow$ that builds function types. It is the canonical and simplest example of a typed lambda calculus. The simply typed lambda calculus was originally introduced by Alonzo Church in 1940 as an attempt to avoid paradoxical uses of the untyped lambda calculus, and it exhibits many desirable and interesting properties.

The term *simple type* is also used to refer to extensions of the simply typed lambda calculus such as products, coproducts or natural numbers (System T) or even full recursion (like PCF). In contrast, systems which introduce polymorphic types (like System F) or dependent types (like the Logical Framework) are not considered *simply typed*. The former are still considered *simple* because the Church encodings of such structures can be done using only $\rightarrow$ and suitable type variables, while polymorphism and dependency cannot.

Syntax

To define the types, we begin by fixing a set of *base types*, B. These are sometimes called *atomic types* or *type constants*. With this fixed, the syntax of types is:

$$\tau = \tau \rightarrow \tau \mid T \text{ where } T \in B.$$

Here, we use σ and τ to range over types. Informally, the *function type* $\sigma \rightarrow \tau$ refers to the set of functions that, given an input of type σ, produce an output of type τ. By convention, $\rightarrow$ associates to the right: we read $\sigma \rightarrow \tau \rightarrow \rho$ as $\sigma \rightarrow (\tau \rightarrow \rho)$.

We also fix a set of *term constants* for the base types. For example, we might assume a base type `nat`, and the term constants could be the natural numbers. In the original presentation, Church used only two base types: *o* for "the type of propositions" and ι for "the type of individuals". The type *o* has no term constants, whereas ι has one term constant. Frequently the calculus with only one base type, usually *o*, is considered.

The syntax of the simply typed lambda calculus is essentially that of the lambda calculus itself. The term syntax used here is as follows:

$e = x \mid \lambda x : \tau.\ e \mid e\ e \mid\ c$ where *c* is a term constant.

That is, *variable reference*, *abstractions*, *application*, and *constant*. A variable reference *x* is *bound* if it is inside of an abstraction binding *x*. A term is *closed* if there are no unbound variables.

Typing rules

To define the set of well typed lambda terms of a given type, we will define a typing relation between terms and types. First, we introduce *typing contexts* $\Gamma, \Delta, \cdots$, which are sets of typing assumptions. A *typing assumption* has the form *x*:σ, meaning *x* has type σ.

The *typing relation* $\Gamma \vdash e : \sigma$ indicates that *e* is a term of type σ in context Γ. It is therefore said that "*e* is *well-typed* (at σ)". Instances of the typing relation are called *typing judgments*. The validity of a typing judgment is shown by providing a *typing derivation*, constructed using the following rules (wherein the premises above the line allow us to derive the conclusion below the line):

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}(1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T}(2)$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.\ e : \sigma \to \tau}(3) \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}(4)$$

In other words,

1. If *x* has type σ in the context, we know that *x* has type σ.
2. Term constants have the appropriate base types.
3. If, in a certain context with *x* having type σ, *e* has type τ, then, in the same context without *x*, $\lambda x : \sigma.\ e$ has type $\sigma \to \tau$.
4. If, in a certain context, e_1 has type $\sigma \to \tau$, and e_2 has type σ, then $e_1\ e_2$ has type τ.

Examples of closed terms, i.e. terms typable in the empty context, are:

- For every type τ, a term $\lambda x : \tau.x : \tau \to \tau$ (the I-combinator/identity function),
- For types σ,τ, a term $\lambda x : \sigma.\lambda y : \tau.x : \sigma \to \tau \to \sigma$ (the K-combinator), and
- For types τ,τ',τ'', a term $\lambda x : \tau \to \tau' \to \tau''.\lambda y : \tau \to \tau'.\lambda z : \tau.xz(yz) : (\tau \to \tau' \to \tau'') \to (\tau \to \tau') \to \tau \to \tau''$
- (the S-combinator).

These are the typed lambda calculus representations of the basic combinators of combinatory logic.

Each type τ is assigned an order, a number $o(\tau)$. For base types, $o(T) = 0$; for function types, $o(\sigma \to \tau) = \max(o(\sigma) + 1, o(\tau))$. That is, the order of a type measures the depth of the most left-nested arrow. Hence:

$$o(\iota \to \iota \to \iota) = 1$$
$$o((\iota \to \iota) \to \iota) = 2$$

Semantics

Intrinsic vs. extrinsic interpretations

Broadly speaking, there are two different ways of assigning meaning to the simply typed lambda calculus, as to typed languages more generally, sometimes called *intrinsic* vs. *extrinsic*, or *Church-style* vs. *Curry-style*. An intrinsic/Church-style semantics only assigns meaning to well-typed terms, or more precisely, assigns meaning directly to typing derivations. This has the effect that terms differing only by type annotations can nonetheless be assigned different meanings. For example, the identity term $\lambda x : \mathtt{int}.\ x$ on integers and the identity term $\lambda x : \mathtt{bool}.\ x$ on booleans may mean different things. In contrast, an extrinsic/Curry-style semantics assigns meaning to terms regardless of typing, as they would be interpreted in an untyped language. In this view, $\lambda x : \mathtt{int}.\ x$ and $\lambda x : \mathtt{bool}.\ x$ mean the same thing (i.e., the same thing as $\lambda x.\ x$).

The distinction between intrinsic and extrinsic semantics is sometimes associated with the presence or absence of annotations on lambda abstractions, but strictly speaking this usage is imprecise. It is possible to define a Curry-style semantics on annotated terms simply by ignoring the types (i.e., through type erasure), as it is possible to give a Church-style semantics on unannotated terms when the types can be deduced from context (i.e., through type inference). The essential difference between intrinsic and extrinsic approaches is just whether the typing rules are viewed as defining the language, or as a formalism for verifying properties of a more primitive underlying language. Most of the different semantic interpretations discussed below can be seen through either Church or Curry goggles.

Equational theory

The simply typed lambda calculus has the same theory of βη-equivalence as untyped lambda calculus, but subject to type restrictions. The equation $(\lambda x : \sigma.t)\, u =_\beta t[x := u]$ holds in context Γ whenever $\Gamma, x : \sigma \vdash t : \tau$ and $\Gamma \vdash u : \sigma$, while the equation $t =_\eta \lambda x : \sigma.t\, x$ holds whenever $\Gamma \vdash t : \sigma \to \tau$.

Operational semantics

Likewise, the operational semantics of simply typed lambda calculus can be fixed as for the untyped lambda calculus, using call by name, call by value, or other evaluation strategies. As for any typed language, type safety is a fundamental property of all of these evaluation strategies. Additionally, the strong normalization property described below implies that any evaluation strategy will terminate on all simply typed terms.

Categorical semantics

The simply typed lambda calculus (with βη-equivalence) is the internal language of Cartesian Closed Categories (CCCs), as was first observed by Lambek.

Proof-theoretic semantics

The simply typed lambda calculus is closely related to the implicational fragment of propositional intuitionistic logic (i.e., minimal logic) via the Curry–Howard isomorphism: terms correspond precisely to proofs in natural deduction, and inhabited types are exactly the tautologies of minimal logic.

Alternative syntaxes

The presentation given above is not the only way of defining the syntax of the simply typed lambda calculus. One alternative is to remove type annotations entirely (so that the syntax is identical to the untyped lambda calculus), while ensuring that terms are well-typed via Hindley-Milner type inference. The inference algorithm is terminating, sound, and complete: whenever a term is typable, the algorithm computes its type. More precisely, it computes the term's principal type, since often an unannotated term (such as $\lambda x.\ x$) may have more than one type ($\texttt{int} \to \texttt{int}$, $\texttt{bool} \to \texttt{bool}$, etc., which are all instances of the principal type $\alpha \to \alpha$).

Another alternative presentation of simply typed lambda calculus is based on **bidirectional type checking**, which requires more type annotations than Hindley-Milner inference but is easier to describe. The type system is divided into two judgments, representing both *checking* and *synthesis*, written $\Gamma \vdash e \Leftarrow \tau$ and $\Gamma \vdash e \Rightarrow \tau$ respectively. Operationally, the three components Γ, *e*, and τ are all *inputs* to the checking judgment $\Gamma \vdash e \Leftarrow \tau$, whereas the synthesis judgment $\Gamma \vdash e \Rightarrow \tau$ only takes Γ and *e* as inputs, producing the type τ as output. These judgments are derived via the following rules:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma} \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c \Rightarrow T}$$

$$\frac{\Gamma, x : \sigma \vdash e \Leftarrow \tau}{\Gamma \vdash \lambda x.\ e \Leftarrow \sigma \to \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \sigma \to \tau \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau}$$

Observe that rules – are nearly identical to rules (1)–(4) above, except for the careful choice of checking or synthesis judgments. These choices can be explained like so:

1. If x:σ is in the context, we can synthesize type σ for x.
2. The types of term constants are fixed and can be synthesized.
3. To check that $\lambda x.\ e$has type $\sigma \to \tau$in some context, we extend the context with x:σ and check that e has type τ.
4. If e_1 synthesizes type $\sigma \to \tau$(in some context), and e_2 checks against type σ (in the same context), then $e_1\ e_2$synthesizes type τ.

Observe that the rules for synthesis are read top-to-bottom, whereas the rules for checking are read bottom-to-top. Note in particular that we do **not** need any annotation on the lambda abstraction in rule , because the type of the bound variable can be deduced from the type at which we check the function. Finally, we explain rules and as follows:

5. To check that e has type τ, it suffices to synthesize type τ.
6. If e checks against type τ, then the explicitly annotated term (e:τ) synthesizes τ.

Because of these last two rules coercing between synthesis and checking, it is easy to see that any well-typed but unannotated term can be checked in the bidirectional system, so long as we insert "enough" type annotations. And in fact, annotations are needed only at β-redexes.

General observations

Given the standard semantics, the simply typed lambda calculus is strongly normalizing: that is, well-typed terms always reduce to a value, i.e., a λ abstraction. This is because recursion is not allowed by the typing rules: it is impossible to find types for fixed-point combinators and the looping term $\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$. Recursion can be added to the language by either having a special operator $\mathtt{fix}_\alpha$of type $(\alpha \to \alpha) \to \alpha$or adding general recursive types, though both eliminate strong normalization.

Since it is strongly normalizing, it is decidable whether or not a simply typed lambda calculus program halts: it does! We can therefore conclude that the language is *not* Turing complete.

Important results

- Tait showed in 1967 that β-reduction is strongly normalizing. As a corollary βη-equivalence is decidable. Statman showed in 1977 that the normalisation problem is not elementary recursive. A purely semantic normalisation proof was given by Berger and Schwichtenberg in 1991.
- The unification problem for βη-equivalence is undecidable. Huet showed in 1973 that 3rd order unification is undecidable and this was improved upon by Baxter in 1978 then by Goldfarb in 1981 by showing that 2nd order unification is already undecidable. Whether higher order matching (unification where only one term contains existential variables) is decidable is still open. [2006: Colin Stirling, Edinburgh, has published a proof-sketch in which he claims that the problem is decidable; however, the complete version of the proof is still unpublished]
- We can encode natural numbers by terms of the type $(o \to o) \to (o \to o)$ (Church numerals). Schwichtenberg showed in 1976 that in $\lambda^{\to}$ exactly the extended polynomials are representable as functions over Church numerals; these are roughly the polynomials closed up under a conditional operator.
- A *full model* of $\lambda^{\to}$ is given by interpreting base types as sets and function types by the set-theoretic function space. Friedman showed in 1975 that this interpretation is complete for βη-equivalence, if the base types are interpreted by infinite sets. Statman showed in 1983 that βη-equivalence is the maximal equivalence which is *typically ambiguous*,i.e. closed under type substitutions (*Statman's Typical Ambiguity Theorem*). A corollary of this is that the *finite model property* holds, i.e. finite sets are sufficient to distinguish terms which are not identified by βη-equivalence.
- Plotkin introduced logical relations in 1973 to characterize the elements of a model which are definable by lambda terms. In 1993 Jung and Tiuryn showed that a general form of logical relation (Kripke logical relations with varying arity) exactly characterizes lambda definability. Plotkin and Statman conjectured that it is decidable whether a given element of a model generated from finite sets is definable by a lambda term (*Plotkin-Statman-conjecture*). The conjecture was shown to be false by Loader in 1993.

Chapter 9

Type Polymorphism and Type System

Type polymorphism

In computer science, **polymorphism** is a programming language feature that allows values of different data types to be handled using a uniform interface. The concept of parametric polymorphism applies to both data types and functions. A function that can evaluate to or be applied to values of different types is known as a *polymorphic function.* A data type that can appear to be of a generalized type (e.g., a list with elements of arbitrary type) is designated *polymorphic data type* like the generalized type from which such specializations are made.

There are two fundamentally different kinds of polymorphism, originally informally described by Christopher Strachey in 1967. If the function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations, it is called **ad-hoc polymorphism**. Ad-hoc polymorphism is supported in many languages using function and method overloading.

If all code is written without mention of any specific type and thus can be used transparently with any number of new types, it is called **parametric polymorphism**. John C. Reynolds (and later Jean-Yves Girard) formally developed this notion of polymorphism as an extension to the lambda calculus (called the polymorphic lambda calculus, or System F). Parametric polymorphism is widely supported in statically typed functional programming languages. In the object-oriented programming community, programming using parametric polymorphism is often called *generic programming*.

In object-oriented programming, **inclusion polymorphism** is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common super class. Inclusion polymorphism is generally supported through subtyping, i.e., objects of different types are entirely substitutable for objects of another type (their base type(s)) and thus can be handled via a common interface. Alternately, inclusion polymorphism may be achieved through type coercion, also known as type casting.

Polymorphism in (early-bound) strongly-typed languages

Parametric polymorphism

Parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. Using **parametric polymorphism**, a function or a data type can be written generically so that it can handle values *identically* without depending on their type. Such functions and data types are called **generic functions** and **generic datatypes** respectively and form the basis of generic programming.

For example, a function `append` that joins two lists can be constructed so that it does not care about the type of elements: it can append lists of integers, lists of real numbers, lists of strings, and so on. Let the *type variable* ***a*** denote the type of elements in the lists. Then `append` can be typed $[a] \times [a] \rightarrow [a]$, where $[a]$ denotes the type of lists with elements of type a. We say that the type of `append` is *parameterized by* ***a*** for all values of a. (Note that since there is only one type variable, the function cannot be applied to just any pair of lists: the pair, as well as the result list, must consist of the same type of elements.) For each place where `append` is applied, a value is decided for a.

Parametric polymorphism was first introduced to programming languages in ML in 1976. Today it exists in Standard ML, OCaml, Ada, Haskell, Visual Prolog and others. Java, C#, Visual Basic .NET and Delphi (CodeGear) have each recently introduced "generics" for parametric polymorphism. Some implementations of type polymorphism are superficially similar to parametric polymorphism while also introducing ad-hoc aspects. One example is C++ template specialization.

The most general form of polymorphism is "higher-rank impredicative polymorphism". Two popular restrictions of this form are restricted rank polymorphism (for example, rank-1 or *prenex* polymorphism) and predicative polymorphism. Together, these restrictions give "predicative prenex polymorphism", which is essentially the form of polymorphism found in ML and early versions of Haskell.

Rank restrictions

Rank-1 (prenex) polymorphism

In a *prenex polymorphic* system, type variables may not be instantiated with polymorphic types. This is very similar to what is called "ML-style" or "Let-polymorphism" (technically ML's Let-polymorphism has a few other syntactic restrictions).

This restriction makes the distinction between polymorphic and non-polymorphic types very important; thus in predicative systems polymorphic types are sometimes referred to as *type schemas* to distinguish them from ordinary (monomorphic) types, which are sometimes called *monotypes*. A consequence is that all types can be written in a form which places all quantifiers at the outermost (prenex) position.

For example, consider the `append` function described above, which has type $[a] \times [a] \to [a]$; in order to apply this function to a pair of lists, a type must be substituted for the variable *a* in the type of the function such that the type of the arguments matches up with the resulting function type. In an *impredicative* system, the type being substituted may be any type whatsoever, including a type that is itself polymorphic; thus `append` can be applied to pairs of lists with elements of any type—even to lists of polymorphic functions such as `append` itself.

Polymorphism in the language ML and its close relatives is predicative. This is because predicativity, together with other restrictions, makes the type system simple enough that type inference is possible. In languages where explicit type annotations are necessary when applying a polymorphic function, the predicativity restriction is less important; thus these languages are generally impredicative. Haskell manages to achieve type inference without predicativity but with a few complications.

*Rank-***k** *polymorphism*

For some fixed value *k*, rank-*k* polymorphism is a system in which a quantifier may not appear to the left of more than *k* arrows (when the type is drawn as a tree).

Type inference for rank-2 polymorphism is decidable, but reconstruction for rank-3 and above is not.

*Rank-***n** *("higher-rank") polymorphism*

Rank-*n* polymorphism is polymorphism in which quantifiers may appear to the left of arbitrarily many arrows.

Predicativity restrictions

Predicative polymorphism

In a predicative parametric polymorphic system, a type τ containing a type variable α may not be used in such a way that α is instantiated to a polymorphic type.

Impredicative polymorphism ("first class" polymorphism)

Also called first-class polymorphism. Impredicative polymorphism allows the instantiation of a variable in a type τ with any type, including polymorphic types, such as τ itself.

In type theory, the most frequently studied impredicative typed λ-calculi are based on those of the lambda cube, especially System F. Predicative type theories include Martin-Löf Type Theory and NuPRL.

Bounded parametric polymorphism

Cardelli and Wegner recognized in 1985 the advantages of allowing *bounds* on the type parameters. Many operations require some knowledge of the data types but can otherwise work parametrically. For example, to check whether an item is included in a list, we need to compare the items for equality. In Standard ML, type parameters of the form *''a* are restricted so that the equality operation is available, thus the function would have the type *''a* × *''a* list → bool and *''a* can only be a type with defined equality. In Haskell, bounding is achieved by requiring types to belong to a type class; thus the same function has the type $Eq\,\alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow Bool$ in Haskell. In most object-oriented programming languages that support parametric polymorphism, parameters can be constrained to be subtypes of a given type.

Subtyping polymorphism (or inclusion polymorphism)

Some languages employ the idea of *subtypes* to restrict the range of types that can be used in a particular case of parametric polymorphism. In these languages, **subtyping polymorphism** (sometimes referred to as dynamic polymorphism) allows a function to be written to take an object of a certain type *T*, but also work correctly if passed an object that belongs to a type *S* that is a subtype of *T* (according to the Liskov substitution principle). This type relation is sometimes written *S* <: *T*. Conversely, *T* is said to be a *supertype* of *S*—written *T* :> *S*.

For example, if `Number`, `Rational`, and `Integer` are types such that `Number` :> `Rational` and `Number` :> `Integer`, a function written to take a `Number` will work equally well when passed an `Integer` or `Rational` as when passed a `Number`. The actual type of the object can be hidden from clients into a black box, and accessed via object identity. In fact, if the `Number` type is *abstract*, it may not even be possible to get your hands on an object whose *most-derived* type is `Number`. This particular kind of type hierarchy is known—especially in the context of the Scheme programming language—as a *numerical tower*, and usually contains many more types.

Object-oriented programming languages offer subtyping polymorphism using *subclassing* (also known as *inheritance*). In typical implementations, each class contains what is called a *virtual table*—a table of functions that implement the polymorphic part of the class interface—and each object contains a pointer to the "vtable" of its class, which is then consulted whenever a polymorphic method is called. This mechanism is an example of:

- *late binding*, because virtual function calls are not bound until the time of invocation, and
- *single dispatch* (i.e., single-argument polymorphism), because virtual function calls are bound simply by looking through the vtable provided by the first argument (the `this` object), so the runtime types of the other arguments are completely irrelevant.

The same goes for most other popular object systems. Some, however, such as CLOS, provide *multiple dispatch*, under which method calls are polymorphic in *all* arguments.

Ad-hoc polymorphism with early binding

Strachey chose the term **ad-hoc polymorphism** to refer to polymorphic functions which can be applied to arguments of different types, but which behave **differently** depending on the type of the argument to which they are applied (also known as **function overloading**). The term "ad hoc" in this context is not intended to be pejorative; it refers simply to the fact that this type of polymorphism is not a fundamental feature of the type system.

Ad-hoc polymorphism is a dispatch mechanism: control moving through one named function is dispatched to various other functions without having to specify the exact function being called. Overloading allows multiple functions taking different types to be defined with the same name; the compiler or interpreter automatically calls the right one. This way, functions appending lists of integers, lists of strings, lists of real numbers, and so on could be written, and all be called *append*—and the right *append* function would be called based on the type of lists being appended. This differs from parametric polymorphism, in which the function would need to be written *generically*, to work with any kind of list. Using overloading, it is possible to have a function perform two completely different things based on the type of input passed to it; this is not possible with parametric polymorphism. Another way to look at overloading is that a routine is uniquely identified not by its name, but by the combination of its name and the number, order and types of its parameters.

This type of polymorphism is common in object-oriented programming languages, many of which allow operators to be overloaded in a manner similar to functions. Some languages which are not dynamically typed and lack ad-hoc polymorphism (including type classes) have longer function names such as `print_int`, `print_string`, etc. This can be seen as advantage (more descriptive) or a disadvantage (overly verbose) depending on one's point of view.

An advantage that is sometimes gained from overloading is the appearance of specialization, e.g., a function with the same name can be implemented in multiple different ways, each optimized for the particular data types that it operates on. This can provide a convenient interface for code that needs to be specialized to multiple situations for performance reasons.

Since overloading is done at compile time, it is not a substitute for late binding as found in subtyping polymorphism.

Ad-hoc polymorphism with late binding

The previous section notwithstanding, there are other ways in which ad-hoc polymorphism can work out. Consider for example the Smalltalk language. In Smalltalk,

the overloading is done at run time because the methods ("function implementation") for each overloaded message ("overloaded function") are resolved when they are about to be executed. This happens at run time, after the program is compiled. Therefore, polymorphism is given by subtyping polymorphism as in other languages, and it is also extended in functionality by ad-hoc polymorphism at run time.

A closer look will also reveal that Smalltalk provides a slightly different variety of ad-hoc polymorphism. Since Smalltalk has a late bound execution model, and since it provides objects the ability to handle messages which are not understood, it is possible to go ahead and implement functionality using polymorphism without explicitly overloading a particular message. This may not be generally recommended practice for everyday programming, but it can be quite useful when implementing proxies.

Also, while in general terms common class method and constructor overloading is not considered polymorphism, there are more uniform languages in which classes are regular objects. In Smalltalk, for instance, classes are regular objects. In turn, this means messages sent to classes can be overloaded, and it is also possible to create objects that behave like classes without their classes inheriting from the hierarchy of classes. These are effective techniques which can be used to take advantage of Smalltalk's powerful reflection capabilities. Similar arrangements are also possible in languages such as Self and Newspeak.

Example

Imagine an operator + that may be used in the following ways:

1. `1 + 2 = 3`
2. `3.14 + 0.0015 = 3.1415`
3. `1 + 3.7 = 4.7`
4. `[1, 2, 3] + [4, 5, 6] = [1, 2, 3, 4, 5, 6]`
5. `[true, false] + [false, true] = [true, false, false, true]`
6. `"sat" + "ish" = "satish"`

Overloading

To handle these six function calls, four different pieces of code are needed—or *three*, if strings are considered to be lists of characters:

- In the first case, integer addition must be invoked.
- In the second and third cases, floating-point addition must be invoked (with type promotion, or type coercion, in the third case).
- In the fourth and fifth cases, list concatenation must be invoked.
- In the last case, string concatenation must be invoked.

Thus, the name + actually refers to three or four completely different functions. This is an example of *overloading*.

Type system

In computer science, a **type system** may be defined as "a tractable syntactic framework for classifying phrases according to the kinds of values they compute". A type system associates *types* with each computed value. By examining the flow of these values, a type system attempts to prove that no *type errors* can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense.

A compiler may use the static type of a value to optimize the storage it needs and the choice of algorithms for operations on the value. In many C compilers the "float" data type, for example, is represented in 32 bits, in accord with the IEEE specification for single-precision floating point numbers. C thus uses floating-point-specific operations on those values (floating-point addition, multiplication, etc.).

The depth of type constraints and the manner of their evaluation affect the *typing* of the language. A programming language may further associate an operation with varying concrete algorithms on each type in the case of type polymorphism. Type theory is the study of type systems, although the concrete type systems of programming languages originate from practical issues of computer architecture, compiler implementation, and language design.

Fundamentals

Assigning data types (*typing*) gives meaning to sequences of bits. Types usually have associations either with values in memory or with objects such as variables. Because any value simply consists of a sequence of bits in a computer, hardware makes no intrinsic distinction even between memory addresses, instruction code, characters, integers and floating-point numbers, being unable to discriminate between them based on bit pattern alone. Associating a sequence of bits and a type informs programs and programmers how that sequence of bits should be understood.

Major functions provided by type systems include:

- *Safety* – Use of types may allow a compiler to detect meaningless or probably invalid code. For example, we can identify an expression `3 / "Hello, World"` as invalid because the rules of arithmetic do not specify how to divide an integer by a string. As discussed below, strong typing offers more safety, but generally does not guarantee complete safety.
- *Optimization* – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of 4 bytes, the compiler may be able to use more efficient machine instructions.

- *Documentation* – In more expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be represented as integers—but if a programmer declares a function as returning a timestamp type rather than merely an integer type, this documents part of the meaning of the function.
- *Abstraction* (or *modularity*) – Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can think of a string as a collection of character values instead of as a mere array of bytes. Or, types can allow programmers to express the interface between two subsystems. This helps localize the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate.

Type safety contributes to program correctness, but cannot guarantee it unless the type checking itself becomes an undecidable problem. Depending on the specific type system, a program may give the wrong result and be safely typed, producing no compiler errors. For instance, division by zero is not caught by the type checker in most programming languages; instead it is a runtime error. To prove the absence of more general defects, other kinds of formal methods, collectively known as program analyses, are in common use, as well as software testing, a widely used empirical method for finding errors that the type checker cannot detect.

A program typically associates each value with one particular type (although a type may have more than one subtype). Other entities, such as objects, modules, communication channels, dependencies, or even types themselves, can become associated with a type. Some implementations might make the following identifications (though these are technically different concepts):

- data type – a type of a value
- class – a type of an object
- kind – a type of a type

A *type system*, specified for each programming language, controls the ways typed programs may behave, and makes behavior outside these rules illegal. An *effect system* typically provides more fine-grained control than does a type system.

Formally, type theory studies type systems. More elaborate type systems (such as dependent types) allow for finer-grained program specifications to be verified by a type checker, but this comes at a price, as type inference and other properties generally become undecidable, and type checking itself is dependent on user-supplied proofs. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in type safe manner. As Mark Manasse concisely put it:

The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs

may not have meanings ascribed to them. The quest for richer type systems results from this tension.

Type checking

The process of verifying and enforcing the constraints of types – *type checking* – may occur either at compile-time (a static check) or run-time (a dynamic check). If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions which do not lose information), one can refer to the process as *strongly typed*, if not, as *weakly typed*. The terms are not used in a strict sense.

Static typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Statically typed languages include Ada, ActionScript 3, C, C++, C#, Eiffel, F#, Go, JADE, Java, Fortran, Haskell, ML, Objective-C, Pascal, Perl (with respect to distinguishing scalars, arrays, hashes and subroutines) and Scala. Static typing is a limited form of program verification: accordingly, it allows many type errors to be caught early in the development cycle. Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed. Program execution may also be made more efficient (i.e. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations.

Because they evaluate type information during compilation, and therefore lack type information that is only available at run-time, static type checkers are conservative. They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed. For example, even if an expression `<complex test>` always evaluates to `true` at run-time, a program containing the code

```
if <complex test> then 42 else <type error>
```

will be rejected as ill-typed, because a static analysis cannot determine that the `else` branch won't be taken. The conservative behaviour of static type checkers is advantageous when `<complex test>` evaluates to `false` infrequently: A static type checker can detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors because the combination of all places where values are created and all places where a certain value is used must be taken into account.

The most widely used statically typed languages are not formally type safe. They have "loopholes" in the programming language specification enabling programmers to write code that circumvents the verification performed by a static type checker and so address a wider range of problems. For example, most C-style languages have type punning, and Haskell has such features as `unsafePerformIO`: such operations may be unsafe at

runtime, in that they can cause unwanted behaviour due to incorrect typing of values when the program runs.

Dynamic typing

A programming language is said to be dynamically typed when the majority of its type checking is performed at run-time as opposed to at compile-time. In dynamic typing, values have types but variables do not; that is, a variable can refer to a value of any type. Dynamically typed languages include Erlang, Groovy, JavaScript, Lisp, Lua, Objective-C, Perl (with respect to user-defined types but not built-in types), PHP, Prolog, Python, Ruby, Smalltalk and Tcl. Compared to static typing, dynamic typing can be more flexible (e.g., by allowing programs to generate types and functionality based on run-time data), though at the expense of fewer *a priori* guarantees. This is because a dynamically typed language accepts and attempts to execute some programs which may be ruled as invalid by a static type checker. The term "dynamic language" means something different ("runtime dynamism") and a dynamic language is not necessarily dynamically typed.

Dynamic typing may result in runtime type errors—that is, at runtime, a value may have an unexpected type, and an operation nonsensical for that type is applied. This operation may occur long after the place where the programming mistake was made—that is, the place where the wrong type of data passed into a place it should not have. This may make the bug difficult to locate.

Dynamically typed language systems' run-time checks can potentially be more sophisticated than those of statically typed languages, as they can use dynamic information as well as any information from the source code. On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and these checks are repeated for every execution of the program.

Development in dynamically typed languages is often supported by programming practices such as unit testing. Testing is a key practice in professional software development, and is particularly important in dynamically typed languages. In practice, the testing done to ensure correct program operation can detect a much wider range of errors than static type-checking, but conversely cannot search as comprehensively for the errors that static type checking is able to detect.

Combinations of dynamic and static typing

The presence of static typing in a programming language does not necessarily imply the absence of all dynamic typing mechanisms. For example, Java, and various other object-oriented languages, while using static typing, require for certain operations (downcasting) the support of runtime type tests, a form of dynamic typing.

Certain languages, for example Clojure, are dynamically typed by default but allow this behaviour can be overridden through the use of explicit type hints that result in static

typing. One reason to use such hints would be to achieve the performance benefits of static typing in performance-sensitive parts of code.

As of the 4.0 Release, the .NET Framework supports a variant of dynamic typing via the System.Dynamic namespace whereby a *static* object of type 'dynamic' is a placeholder for the .NET runtime to interrogate its dynamic facilities to resolve the object reference.

Static and dynamic type checking in practice

The choice between static and dynamic typing requires trade-offs.

Static typing can find type errors reliably at compile time. This should increase the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, and thus what proportion of those bugs which are written would be caught by static typing. Static typing advocates believe programs are more reliable when they have been well type-checked, while dynamic typing advocates point to distributed code that has proven reliable and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased. Advocates of dependently typed languages such as Dependent ML and Epigram have suggested that almost all bugs can be considered type errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.

Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce optimized machine code. Further, compilers for statically typed languages can find assembler shortcuts more easily. Some dynamically typed languages such as Common Lisp allow optional type declarations for optimization for this very reason. Static typing makes this pervasive.

By contrast, dynamic typing may allow compilers to run more quickly and allow interpreters to dynamically load new code, since changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit. This too may reduce the edit-compile-test-debug cycle.

Statically typed languages which lack type inference (such as Java and C) require that programmers declare the types they intend a method or function to use. This can serve as additional documentation for the program, which the compiler will not permit the programmer to ignore or permit to drift out of synchronization. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala and to a lesser extent C#), so this is not a necessary consequence of static typing.

Dynamic typing allows constructs that some static type checking would reject as illegal. For example, *eval* functions, which execute arbitrary data as code, become possible. Furthermore, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full-fledged data structure (usually for the purposes of experimentation and testing).

Dynamic typing is used in duck typing which can support easier code reuse.

Dynamic typing typically makes metaprogramming more effective and easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code. More advanced run-time constructs such as metaclasses and introspection are often more difficult to use in statically typed languages.

Strong and weak typing

One definition of *strongly typed* involves preventing success for an operation on arguments which have the wrong type. A C cast gone wrong exemplifies the problem of absent strong typing; if a programmer casts a value from one type to another in C, not only must the compiler allow the code at compile time, but the runtime must allow it as well. This may permit more compact and faster C code, but it can make debugging more difficult.

Weak typing means that a language implicitly converts (or casts) types when used. For example, we may have:

```
var x := 5;     // (1)  (x is an integer)
var y := "37"; // (2)  (y is a string)
x + y;          // (3)  (?)
```

In a weakly typed language, the result of this operation is unclear. Some languages, such as Visual Basic, would produce runnable code producing the result 42: the system would convert the string "37" into the number 37 to forcibly make sense of the operation. Other languages like JavaScript would produce the result "537": the system would convert the number 5 to the string "5" and then concatenate the two. In both Visual Basic and JavaScript, the resulting type is determined by rules that take both operands into consideration. In some languages, such as AppleScript, the type of the resulting value is determined by the type of the left-most operand only.

Safely and unsafely typed systems

A third way of categorizing the type system of a programming language uses the safety of typed operations and conversions. Computer scientists consider a language "type-safe" if it does not allow operations or conversions which lead to erroneous conditions.

Some observers use the term *memory-safe language* (or just *safe language*) to describe languages that do not allow undefined operations to occur. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors.

```
var x := 5;      // (1)
var y := "37";  // (2)
var z := x + y; // (3)
```

In languages like Visual Basic, variable z in the example acquires the value 42. While the programmer may or may not have intended this, the language defines the result specifically, and the program does not crash or assign an ill-defined value to z. In this respect, such languages are type-safe; however, if the value of y was a string that could not be converted to a number (e.g. "hello world"), the results would be undefined. Such languages are type-safe (in that they will not crash) but can easily produce undesirable results.

Now let us look at the same example in C:

```
int x = 5;
char y[] = "37";
char* z = x + y;
```

In this example z will point to a memory address five characters beyond y, equivalent to three characters after the terminating zero character of the string pointed to by y. The content of that location is undefined, and might lie outside addressable memory. The mere computation of such a pointer may result in undefined behavior (including the program crashing) according to C standards, and in typical systems dereferencing z at this point could cause the program to crash. We have a well-typed, but not memory-safe program—a condition that cannot occur in a type-safe language.

Polymorphism and types

The term "polymorphism" refers to the ability of code (in particular, methods or classes) to act on values of multiple types, or to the ability of different instances of the same data-structure to contain elements of different types. Type systems that allow polymorphism generally do so in order to improve the potential for code re-use: in a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once, rather than once for each type of element with which they plan to use it. For this reason computer scientists sometimes call the use of certain forms of polymorphism *generic programming*. The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping.

Duck typing

In "duck typing", a statement calling a method *m* on an object does not rely on the declared type of the object; only that the object, of whatever type, must implement the method called. One way of looking at this is that in duck typing systems the type of an object is intrinsic to the object and is determined by what methods it implements, and hence that a duck typing system is by definition type-safe since one can only invoke operations an object actually implements. Another way of looking at this is that the object is a member of *several* types, including a type that describes the fact that it "has a method *m*." Type checking however occurs only on demand at runtime, every time the method *m* needs to be executed, not at compile-time or load-time.

Duck typing differs from structural typing in that, if the *part* (of the whole module structure) needed for a given local computation is present *at runtime*, the duck type system is satisfied in its type identity analysis. On the other hand, a structural type system would require the analysis of the whole module structure at compile-time to determine type identity or type dependence.

Duck typing differs from a nominative type system in a number of aspects. The most prominent ones are that, for duck typing, type information is determined at runtime (as contrasted to compile-time) and the name of the type is irrelevant to determine type identity or type dependence; only partial structure information is required for that, for a given point in the program execution.

Initially coined by Alex Martelli in the Python community, duck typing uses the premise that (referring to a value) "if it walks like a duck, and quacks like a duck, then it is a duck".

Specialized type systems

Many type systems have been created that are specialized for use in certain environments, with certain types of data, or for out-of-band static program analysis. Frequently these are based on ideas from formal type theory and are only available as part of prototype research systems.

Dependent types

Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. For example, "matrix(3,3)" might be the type of a 3×3 matrix. We can then define typing rules such as the following rule for matrix multiplication:

$$\text{matrix\_multiply} : \text{matrix}(k,m) \times \text{matrix}(m,n) \rightarrow \text{matrix}(k,n)$$

where k, m, n are arbitrary positive integer values. A variant of ML called Dependent ML has been created based on this type system, but because type-checking conventional dependent types is undecidable, not all programs using them can be type-checked without some kind of limits. Dependent ML limits the sort of equality it can decide to Presburger arithmetic. Other languages such as Epigram make the value of all expressions in the language decidable so that type checking can be decidable, it is also possible to make the language Turing complete at the price of undecidable type checking like in Cayenne.

Linear types

Linear types, based on the theory of linear logic, and closely related to uniqueness types, are types assigned to values having the property that they have one and only one reference to them at all times. These are valuable for describing large immutable values such as strings, files, and so on, because any operation that simultaneously destroys a

linear object and creates a similar object (such as '`str = str + "a"`') can be optimized "under the hood" into an in-place mutation. Normally this is not possible because such mutations could cause side effects on parts of the program holding other references to the object, violating referential transparency. They are also used in the prototype operating system Singularity for interprocess communication, statically ensuring that processes cannot share objects in shared memory in order to prevent race conditions. The Clean language (a Haskell-like language) uses this type system in order to gain a lot of speed while remaining safe.

Intersection types

Intersection types are types describing values that belong to *both* of two other given types with overlapping value sets. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting *either* signed or unsigned chars, because it is compatible with both types.

Intersection types are useful for describing overloaded function types: For example, if "int → int" is the type of functions taking an integer argument and returning an integer, and "float → float" is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "int → int" function safely; it simply would not use the "float → float" functionality.

In a subclassing hierarchy, the intersection of a type and an ancestor type (such as its parent) is the most derived type. The intersection of sibling types is empty.

The Forsythe language includes a general implementation of intersection types. A restricted form is refinement types.

Union types

Union types are types describing values that belong to *either* of two types. For example, in C, the signed char has range -128 to 127, and the unsigned char has range 0 to 255, so the union of these two types would have range -128 to 255. Any function handling this union type would have to deal with integers in this complete range. More generally, the only valid operations on a union type are operations that are valid on *both* types being unioned. C's "union" concept is similar to union types, but is not typesafe because it permits operations that are valid on *either* type, rather than *both*. Union types are important in program analysis, where they are used to represent symbolic values whose exact nature (e.g., value or type) is not known.

In a subclassing hierarchy, the union of a type and an ancestor type (such as its parent) is the ancestor type. The union of sibling types is a subtype of their common ancestor (that

is, all operations permitted on their common ancestor are permitted on the union type, but they may also have other valid operations in common).

Existential types

Existential types are frequently used in connection with record types to represent modules and abstract data types, due to their ability to separate implementation from interface. For example, the type "T = $\exists$ X { a: X; f: (X → int); }" describes a module interface that has a data member of type *X* and a function that takes a parameter of the *same* type *X* and returns an integer. This could be implemented in different ways; for example:

- intT = { a: int; f: (int → int); }
- floatT = { a: float; f: (float → int); }

These types are both subtypes of the more general existential type T and correspond to concrete implementation types, so any value of one of these types is a value of type T. Given a value "t" of type "T", we know that "t.f(t.a)" is well-typed, regardless of what the abstract type *X* is. This gives flexibility for choosing types suited to a particular implementation while clients that use only values of the interface type—the existential type—are isolated from these choices.

In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example intT { a: int; f: (int → int); } could also have the type $\exists$ X { a: X; f: (int → int); }. The simplest solution is to annotate every module with its intended type, e.g.:

- intT = { a: int; f: (int → int); } **as** $\exists$ X { a: X; f: (X → int); }

Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type". The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification.

Explicit or implicit declaration and inference

Many static type systems, such as those of C and Java, require *type declarations*: The programmer must explicitly associate each variable with a particular type. Others, such as Haskell's, perform *type inference*: The compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function *f(x,y)* which adds *x* and *y* together, the compiler can infer that *x* and *y* must be numbers – since addition is only defined for numbers. Therefore, any call to *f* elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.

Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression `3.14` might imply a type of floating-point, while `[1, 2, 3]` might imply a list of integers – typically an array.

Type inference is in general possible if it is decidable in the type theory in question. Moreover, even if inference is undecidable in general for a given type theory, inference is often possible for a large subset of real-world programs. Haskell's type system, a version of Hindley-Milner, is a restriction of System Fω to so-called rank-1 polymorphic types, in which type inference is decidable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference undecidable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.)

Chapter 10

Algebraic Data Type and Calculus of Constructions

Algebraic data type

In computer programming, particularly functional programming and type theory, an **algebraic data type** (sometimes also called a *variant type*) is a datatype each of whose values is data from other datatypes wrapped in one of the constructors of the datatype. Any wrapped datum is an argument to the constructor. In contrast to other datatypes, the constructor is not executed and the only way to operate on the data is to unwrap the constructor using pattern matching.

The most common algebraic data type is a list with two constructors: `Nil` or `[]` for an empty list, and `Cons` (an abbreviation of *cons*truct), `::`, or `:` for the combination of a new element with a shorter list (for example `Cons 1 [2, 3, 4]` or `1:[2,3,4]`).

Special cases of algebraic types are product types i.e. tuples and records (only one constructor), sum types or tagged unions (many constructors with a single argument) and enumerated types (many constructors with no arguments). Algebraic types are one kind of composite type (i.e. a type formed by combining other types).

An algebraic data type may also be an abstract data type (ADT) if it is exported from a module without its constructors. Values of such a type can only be manipulated using functions defined in the same module as the type itself.

In set theory the equivalent of an algebraic data type is a disjoint union – a set whose elements are pairs consisting of a tag (equivalent to a constructor) and an object of a type corresponding to the tag (equivalent to the constructor arguments).

Examples

For example, in Haskell we can define a new algebraic data type, `Tree`:

```
data Tree = Empty
          | Leaf Int
```

```
          | Node Tree Tree
```

Here, `Empty`, `Leaf` and `Node` are called **data constructors**. `Tree` is a **type constructor** (in this case a nullary one). In the rest of the chapter constructor shall mean data constructor. Similarly, in OCaml syntax the above example may be written:

```
type tree = Empty
          | Leaf of int
          | Node of tree * tree
```

In most languages that support algebraic data types, it's possible to define polymorphic types. Examples are given later here.

Somewhat similar to a function, a data constructor is applied to arguments of an appropriate type, yielding an instance of the data type to which the type constructor belongs. For instance, the data constructor `Leaf` is logically a function `Int -> Tree`, meaning that giving an integer as an argument to `Leaf` produces a value of the type `Tree`. As `Node` takes two arguments of the type `Tree` itself, the datatype is recursive.

Operations on algebraic data types can be defined by using pattern matching to retrieve the arguments. For example, consider a function to find the depth of a `Tree`, given here in Haskell:

```
depth :: Tree -> Int
depth Empty = 0
depth (Leaf n) = 1
depth (Node l r) = 1 + max (depth l) (depth r)
```

Thus, a `Tree` given to `depth` can be constructed using any of `Empty`, `Leaf` or `Node` and we must match for any of them respectively to deal with all cases. In case of `Node`, the pattern extracts the subtrees `l` and `r` for further processing.

Algebraic data types are particularly well-suited to the implementation of abstract syntax. For instance, the following algebraic data type describes a simple language representing numerical expressions:

```
data Expression = Number Int
                | Add Expression Expression
                | Minus Expression
                | Mult Expression Expression
                | Divide Expression Expression
```

An element of such a data type would have a form such as `Mult (Add (Number 4) (Minus (Number 1))) (Number 2)`.

Writing an evaluation function for this language is a simple exercise; however, more complex transformations also become feasible. For instance, an optimization pass in a

compiler might be written as a function taking an abstract expression as input and returning an optimized form.

Explanation

What is happening is that we have a datatype, which can be “one of several types of things.” Each “type of thing” is associated with an identifier called a *constructor*, which can be thought of as a kind of tag for that kind of data. Each constructor can carry with it a different type of data. A constructor could carry no data at all (e.g. "Empty" in the example above), carry one piece of data (e.g. “Leaf” has one Int value), or multiple pieces of data (e.g. “Node” has two Tree values).

When we want to do something with a value of this Tree algebraic data type, we *deconstruct* it using a process known as *pattern matching*. It involves *matching* the data with a series of *patterns*. The example function "depth" above pattern-matches its argument with three patterns. When the function is called, it finds the first pattern that matches its argument, performs any variable bindings that are found in the pattern, and evaluates the expression corresponding to the pattern.

Each pattern has a form that resembles the structure of some possible value of this datatype. The first pattern above simply matches values of the constructor Empty. The second pattern above matches values of the constructor Leaf. Patterns are recursive, so then the data that is associated with that constructor is matched with the pattern "n". In this case, a lowercase identifier represents a pattern that matches any value, which then is bound to a variable of that name — in this case, a variable “`n`” is bound to the integer value stored in the data type — to be used in the expression to be evaluated.

The recursion in patterns in this example are trivial, but a possible more complex recursive pattern would be something like `Node (Node (Leaf 4) x) (Node y (Node Empty z))`. Recursive patterns several layers deep are used for example in balancing red-black trees, which involve cases that require looking at colors several layers deep.

The example above is operationally equivalent to the following pseudocode:

```
switch on (data.constructor)
  case Empty:
    return 0
  case Leaf:
    let n = data.field1
    return 1
  case Node:
    let l = data.field1
    let r = data.field2
    return 1 + max (depth l) (depth r)
```

The comparison of this with pattern matching will point out some of the advantages of algebraic data types and pattern matching. First is type safety. The pseudocode above relies on the diligence of the programmer to not access `field2` when the constructor is a

Leaf, for example. Also, the type of `field1` is different for Leaf and Node (for Leaf it is `Int`; for Node it is `Tree`), so the type system would have difficulties assigning a static type to it in a safe way in a traditional record data structure. However, in pattern matching, the type of each extracted value is checked based on the types declared by the relevant constructor, and how many values you can extract is known based on the constructor, so it does not face these problems.

Second, in pattern matching, the compiler statically checks that all cases are handled. If one of the cases of the "depth" function above were missing, the compiler would issue a warning, indicating that a case is not handled. This task may seem easy for the simple patterns above, but with many complicated recursive patterns, the task becomes difficult for the average human (or compiler, if it has to check arbitrary nested if-else constructs) to handle. Similarly, there may be patterns which never match (i.e. it is already covered by previous patterns), and the compiler can also check and issue warnings for these, as they may indicate an error in reasoning.

Do not confuse these patterns with regular expression patterns used in string pattern matching. The purpose is similar — to check whether a piece of data matches certain constraints, and if so, extract relevant parts of it for processing — but the mechanism is very different. This kind of pattern matching on algebraic data types matches on the structural properties of an object rather than on the character sequence of strings.

Theory

A general algebraic data type is a possibly recursive sum type of product types. Each constructor tags a product type to separate it from others, or if there is only one constructor, the data type is a product type. Further, the parameter types of a constructor are the factors of the product type. A parameterless constructor corresponds to the empty product. If a datatype is recursive, the entire sum of products is wrapped in a recursive type, and each constructor also rolls the datatype into the recursive type.

For example, the Haskell datatype:

```
data List a = Nil | Cons a (List a)
```

is represented in type theory as $\lambda\alpha.\mu\beta.1 + \alpha \times \beta$ with constructors $\mathrm{nil}_\alpha = \mathrm{roll}\ (\mathrm{inl}\ \langle\rangle)$ and $\mathrm{cons}_\alpha\ x\ l = \mathrm{roll}\ (\mathrm{inr}\ \langle x, l\rangle)$.

The Haskell List datatype can also be represented in type theory in a slightly different form, as follows: $\mu\phi.\lambda\alpha.1 + \alpha \times \phi\ \alpha$. (Note how the μ and λ constructs are reversed relative to the original.) The original formation specified a type function whose body was a recursive type; the revised version specifies a recursive function on types. (We use the type variable φ to suggest a function rather than a "base type" like β, since φ is like a Greek "f".) Note that we must also now apply the function φ to its argument type α in the body of the type.

For the purposes of the List example, these two formulations are not significantly different; but the second form allows one to express so-called nested data types, i.e., those where the recursive type differs parametrically from the original.

Calculus of constructions

The **calculus of constructions** (**CoC**) is a formal language in which both computer programs and mathematical proofs can be expressed. This language forms the basis of theory behind the Coq proof assistant, which implements the derivative calculus of inductive constructions.

General traits

The CoC is a higher-order typed lambda calculus, initially developed by Thierry Coquand, where types are first-class values. It is thus possible, within the CoC, to define functions from, say, integers to types, types to types as well as functions from integers to integers. Within Barendregt's lambda cube, it is therefore the richest calculus.

The CoC is strongly normalizing, though, by Gödel's incompleteness theorem, it is impossible to prove this property within the CoC since it implies consistency.

The CoC was the basis of the early versions of the Coq proof assistant; later versions were built upon the calculus of inductive constructions, an extension of CoC with native support for inductive datatypes. In the original CoC, inductive datatypes had to be emulated as their polymorphic destructor function.

The basics of the calculus of constructions

The Calculus of Constructions can be considered an extension of the Curry–Howard isomorphism. The Curry–Howard isomorphism associates a term in the simply typed lambda calculus with each natural-deduction proof in intuitionistic propositional logic. The Calculus of Constructions extends this isomorphism to proofs in the full intuitionistic predicate calculus, which includes proofs of quantified statements (which we will also call "propositions").

Terms

A *term* in the calculus of constructions is constructed using the following rules:

- **T** is a term (also called *Type*)
- **P** is a term (also called *Prop*, the type of all propositions)
- Variables (*x*, *y*, ...) are terms

- If *A* and *B* are terms, then so are
  - (AB)
  - $(\lambda x : A.B)$
  - $(\forall x : A.B)$

The calculus of constructions has five kinds of objects:

1. *proofs*, which are terms whose types are *propositions*
2. *propositions*, which are also known as *small types*
3. *predicates*, which are functions that return propositions
4. *large types*, which are the types of predicates. (**P** is an example of a large type)
5. **T** itself, which is the type of large types.

Judgments

In the calculus of constructions, a **judgment** is a typing inference:

$$x_1 : A_1, x_2 : A_2, \ldots \vdash t : B$$

Which can be read as the implication

If variables $x_1, x_2, \cdots$ have types $A_1, A_2, \cdots$, then term *t* has type *B*.

The valid judgments for the calculus of constructions are derivable from a set of inference rules. In the following, we use Γ to mean a sequence of type assignments $x_1 : A_1, x_2 : A_2, \cdots$, and we use **K** to mean either **P** or **T**. We will write *A*:*B*:*C* to mean "*A* has type *B*, and *B* has type *C*". We will write *B*(*x*: = *N*) to mean the result of substituting the term *N* for the variable *x* in the term *B*.

An inference rule is written in the form

$$\frac{\Gamma \vdash A : B}{\Gamma' \vdash C : D}$$

which means

If $\Gamma \vdash A : B$ is a valid judgment, then so is $\Gamma' \vdash C : D$

Inference rules for the calculus of constructions

1. $\dfrac{}{\Gamma \vdash P : T}$

2. $\dfrac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}$

3. $$\frac{\Gamma, x : A \vdash t : B : K}{\Gamma \vdash (\lambda x : A.t) : (\forall x : A.B) : K}$$

4. $$\frac{\Gamma \vdash M : (\forall x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B(x := N)}$$

5. $$\frac{\Gamma \vdash M : A \qquad A =_\beta B \qquad B : K}{\Gamma \vdash M : B}$$

Defining logical operators

The calculus of constructions has very few basic operators: the only logical operator for forming propositions is $\forall$. However, this one operator is sufficient to define all the other logical operators:

$$\begin{array}{rcll} A \Rightarrow B & \equiv & \forall x : A.B & (x \notin B) \\ A \wedge B & \equiv & \forall C : P.(A \Rightarrow B \Rightarrow C) \Rightarrow C & \\ A \vee B & \equiv & \forall C : P.(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C & \\ \neg A & \equiv & \forall C : P.(A \Rightarrow C) & \\ \exists x : A.B & \equiv & \forall C : P.(\forall x : A.(B \Rightarrow C)) \Rightarrow C & \end{array}$$

Defining data types

The basic data types used in computer science can be defined within the Calculus of Constructions:

Booleans

$$\forall A : P.A \Rightarrow A \Rightarrow A$$

Naturals

$$\forall A : P.(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$$

Product $A \times B$

$$A \wedge B$$

Disjoint union $A + B$

$$A \vee B$$

Chapter 11

Composite Data Type and Covariance & Contravariance (Computer Science)

Composite data type

In computer science, **composite data types** are data types which can be constructed in a program using its programming language's primitive data types and other composite types. The act of constructing a composite type is known as composition.

C/C++ structures and classes

A `struct` is C's and C++'s notion of a composite type, a datatype that composes a fixed set of labeled **fields** or **members**. It is so called because of the `struct` keyword used in declaring them, which is short for *structure* or, more precisely, *user-defined data structure*.

In C++, the only difference between a `struct` and a class is the default access level, which is *private* for classes and *public* for `struct`s.

Note that while classes and the `class` keyword were completely new in C++, the C programming language already had a crude type of `struct`s. For all intents and purposes, C++ `struct`s form a superset of C `struct`s: virtually all valid C `struct`s are valid C++ `struct`s with the same semantics.

Declaration

A `struct` declaration consists of a list of fields, each of which can have any type. The total storage required for a `struct` object is the sum of the storage requirements of all the fields, plus any internal padding.

For example:

```
struct Account {
   int account_number;
   char *first_name;
   char *last_name;
   float balance;
};
```

defines a type, referred to as `struct Account`. To create a new variable of this type, we can write `struct Account myAccount;` which has an integer component, accessed by `myAccount.account_number`, and a floating-point component, accessed by `myAccount.balance`, as well as the `first_name` and `last_name` components. The structure `myAccount` contains all four values, and all four fields may be changed independently.

Since writing `struct Account` repeatedly in code becomes cumbersome, it is not unusual to see a `typedef` statement in C code to provide a more convenient synonym for the `struct`.

For example:

```
typedef struct Account_ {
   int    account_number;
   char   *first_name;
   char   *last_name;
   float  balance;
} Account;
```

In C++ code, the `typedef` is not needed because types defined using `struct` are already part of the regular namespace, so the type can be referred to as either `struct Account` or simply `Account`.

As another example, a three-dimensional Vector composite type that uses the floating point data type could be created with:

```
struct Vector {
  float x;
  float y;
  float z;
};
```

A variable named `velocity` with a `Vector` composite type would be declared as `Vector velocity;` Members of the `velocity` would be accessed using a dot notation. For example, `velocity.x = 5;` would set the `x` component of `velocity` equal to 5.

Likewise, a color structure could be created using:

```
struct Color {
```

```
  unsigned int red;
  unsigned int green;
  unsigned int blue;
};
```

In 3D graphics, you usually must keep track of both the position and color of each vertex. One way to do this would be to create a `Vertex` composite type, using the previously created `Vector` and `Color` composite types:

```
struct Vertex {
  Vector position;
  Color color;
};
```

Instantiation

Create a variable of type `Vertex` using the same format as before: `Vertex v;`

Member access

Assign values to the components of `v` like so:

```
v.position.x = 0.0;
v.position.y = 1.5;
v.position.z = 0.0;
v.color.red = 128;
v.color.green = 0;
v.color.blue = 255;
```

Primitive subtype

The primary use of `struct` is for the construction of complex datatypes, but sometimes it is used to create primitive structural subtyping. For example, since Standard C requires that if two structs have the same initial fields, those fields will be represented in the same way, the code

```
struct ifoo_old_stub {
   long x, y;
};
struct ifoo_version_42 {
   long x, y, z;
   char *name;
   long a, b, c;
};
void operate_on_ifoo(struct ifoo_old_stub *);
struct ifoo_version_42 s;
. . .
operate_on_ifoo(&s);
```

will work correctly.

Function types

Function types (or type signatures) are constructed from primitive and composite types, and can serve as types themselves when constructing composite types:

```
typedef struct {
    int x;
    int y;
} Point;

typedef double (*Metric) (Point p1, Point p2);

typedef struct {
    Point centre;
    double radius;
    Metric metric;
} Circle;
```

Covariance and contravariance (computer science)

Within the type system of a programming language, covariance and contravariance refers to the ordering of types from narrower to wider and their interchangeability or equivalence in certain situations (such as parameters, generics, and return types).

- **covariant:** converting from wider (double) to narrower (float).
- **contravariant:** converting from narrower (float) to wider (double).
- **invariant:** Not able to convert (Null).

For example, a type that may assume the values {a,b,c,d} is wider than one that may only assume the values {a,b}. Hence, a type conversion from {a,b}->{a,b,c,d}, such as in the case of passing a float to a function expecting a double, is a covariant conversion. Similarly, a type conversion from {a,b,c,d}->{a,b}, such as in the case of calling a function returning a float in the place of one returning a double, is a contravariant conversion of the function (the function type is its result type).

Note, types may share values yet not be equivalent. For example, types assuming values {a,b} and {b,c}, respectively, are not equivalent to each other...though they are equivalent to a type assuming the values {b} since {b}->{a,b} and {b}->{b,c}.

A class's type equivalence is implied by its inheritance hierarchy (and is the justification for inheritance). However, since all possible changes to a derived class can destroy this

assertion, some languages restrict the presumption of this implicit equivalence in some situations. For example, in C# 3.0 generic parameters did not support co or contravariance; IEnumerable<TypeDerivedFromA> was not assignable to IEnumerable<A> as one might intuit; however, this is now supported in C# 4.0, though standard arrays have always supported covariance & contravariance since .NET was introduced (it doesn't statically guarantee that array assignments are valid and an exception may be thrown at runtime).

Formal Definition

Within the type system of a programming language, a typing rule or a type conversion operator is:

- **covariant** if it preserves the ordering, ≤, of types, which orders types from more specific to more generic;
- **contravariant** if it reverses this ordering, which orders types from more generic to more specific;
- **invariant** if neither of these applies.

These terms come from category theory, which has a general definition of covariance and contravariance that unifies the computer science definition of these terms with the definition used in vector spaces.

This distinction is important in considering argument and return types of methods in class hierarchies. In object-oriented languages such as C++, if class *B* is a subtype of class *A*, then all member functions of *B* must return the same or narrower set of types as *A*; the return type is said to be covariant. On the other hand, if the member functions of *B* take the same or broader set of arguments compared with the member functions of *A*; the argument type is said to be contravariant. The problem for instances of B is how to be perfectly substitutable for instances of A. The only way to guarantee type safety and substitutability is to be equally or more liberal than A on inputs, and to be equally or more strict than A on outputs. Note that not all programming languages guarantee both properties in every context, and that some are unnecessarily strict; they are said not to support covariance or contravariance *in a given context*; the behavior of some programming languages is discussed below.

Typical examples:

- The operator which constructs array types from element types is usually covariant on the base type: since *String* ≤ *Object* then *ArrayOf(String)* ≤ *ArrayOf(Object)*. Note that this is only correct (i.e. type safe) if the array is immutable; if insert and remove operators are permitted, then the insert operator is covariant (e.g. one can insert a *String* into an *ArrayOf(Object)*) and the remove operator is contravariant (e.g. one can remove an *Object* from an *ArrayOf(String)*). Since the mutators have conflicting variance, mutable arrays should be *invariant* on the base type.

- A call to a function with a parameter of type T (defined as *fun f* (x : T) : *Integer*) can be replaced by a call to a function g (defined as *fun g* (x : S) : *Integer*) if $T \leq S$. In other words, if g *cares less* about the type of its parameter, then it can replace f anywhere, since both return an *Integer*. So, in a language accepting function arguments, $g \leq f$ and the type of the parameter to f is said to be contravariant.
- In the general case, the type of the result is covariant.

In object-oriented programming, substitution is also implicitly invoked by *overriding* methods in subclasses: the new method can be used where the old method was invoked in the original code. Programming languages vary widely on their allowed forms of overriding, and on the variance of overridden methods' types.

Origin of the terms

The origin of these terms is in category theory, where the types in the type system form a category C, with arrows representing the subtype relationship. The subtype relationship supposedly reflects the substitution principle: that any expression of type t can be substituted by an expression of type s if $s \leq t$.

Defining a function that accepts type p and returns type r creates a new type $p \to r$ in the type system which the new function name is associated with. This *function definition* operator is actually a functor $F : C \times C \to C$ that creates the said type. From the substitution principle above, this functor must be contravariant in the first argument and covariant in the second.

Need for covariant argument types?

In many strictly-typed languages (with the notable exception of Eiffel, see below), subclassing must allow for substitution. That is, a child class can always stand in for a parent class. This places restrictions on the sorts of relationships that subclassing can represent. In particular, it means that arguments to member functions can only be contravariant and return types can only be covariant, as explained in previous section.

This creates problems in some situations, where argument types should be covariant to model real-life requirements. Suppose you have a class representing a person. A person can see the doctor, so this class might have a method `virtual void Person::see(Doctor d)`. Now suppose you want to make a subclass of the `Person` class, `Child`. That is, a `Child` is a `Person`. One might then like to make a subclass of `Doctor`, `Pediatrician`. If children only visit pediatricians, we would like to enforce that in the type system. However, a naive implementation fails: because a `Child` is a `Person`, `Child::see(d)` must take any `Doctor`, not just a `Pediatrician`.

We could try moving the `see()` method to the `Doctor` class hierarchy, but we would have the same problem: If a `Doctor` could see a `Person` and a `Child` is a `Person`, then

there is still no way to enforce that a `Child` must see a `Pediatrician` and that a `Person` who is not a `Child` cannot see a `Pediatrician` and must see another `Doctor`.

In this case, the visitor pattern could be used to enforce this relationship. Another way to solve the problems, in C++, is using generic programming (see below).

Avoiding the need for covariant argument types

The problem arises since different object oriented languages have different strategies to select the actual code used in a particular context and the first parameter is the object itself (which is *not* contravariant).

However, Castagna showed that all depends on the correct method fetching algorithm: types used for runtime selection of the right method are covariant; types not used for runtime selection of the method are contravariant. In Castagna's work, examples which would suggest the usage of covariance for parameter types are treated with the usage of multiple dispatch, i.e. overriding where the right method is selected also based on the type of some arguments; applying the rule, covariance is allowed for those argument types. However, this solution cannot be applied to most programming languages, since they do not support multiple dispatch.

Note that for (static) overload resolution, the opposite rule applies: types used for compile-time method selection (i.e. parameter types) are contravariant; types not used to select the method are covariant.

These terms are also used in the context of modern programming languages that offer other *functors* to create new types with type variables, e.g., generic programming or parametric polymorphism, and exception handling where method definitions are enriched with annotations that indicate possible failures.

Overview of covariance/contravariance in some programming languages

Both the subtype and method overriding concepts are defined differently between programming languages. They do not necessarily follow the substitution principle above, sometimes adding runtime checking instead. What follows is a simple comparison of how overriding methods behave in some common programming languages.

C++

C++ supports covariant return types in overridden virtual functions. Adding the covariant return type was the first modification of the C++ language approved by the standards committee in 1998.

Arrays in C# and Java

In the above discussion we have shown that type safety requires invariance of array types. However, arrays of reference types are covariant in both languages, and this leads to lack of static type safety: for instance, in C# `string[]` is a subtype of `object[]`, and in Java `String[]` is a subtype of `Object[]`, although with some caveats. For instance, in C#, we have:

```
// a is a single-element array of System.String
string[] a = new string;

// b is an array of System.Object
object[] b = a;

// Assign an integer to b. This would be possible if b really were
// an array of objects, but since it really is an array of strings,
// we will get an ArrayTypeMismatchException with the following
message:
// "Attempted to store an element of the incorrect type into the
array".
b[0] = 1;
```

The same problem exists in Java, too:

```
// a is a single-element array of String
String[] a = new String;

// b is an array of Object
Object[] b = a;

// Assign an Integer to b. This would be possible if b really were
// an array of Object, but since it really is an array of String,
// we will get a java.lang.ArrayStoreException.
b[0] = 1;
```

Note: In the above cases you can **read** from b without problem. It is only when trying to **write** to the array that you must know its real type.

Note that `int[]` is not a subtype of `double[]`.

C#

In C# it is possible to store an object which is an instance of an equal or smaller type in that storage location.

```
object str = "string";
```

Ever since C# 1.0, arrays where the element type is a reference type are covariant.

```
object[] strings = new string;
```

Method group to delegate conversions are covariant in return types, and contravariant in argument types.

Generic delegate types are always invariant in C# 3.0

A variant interface which inherits from another variant interface must do so in a manner which does not introduce problems in the type system

C# 4.0 allows declaration of covariance and contravariance on parameterized interface and delegate types

D

The D Programming Language supports covariance for method overriding:

```
interface IFactory {
    Object Create();
}

class X { }

class XFactory : IFactory {
    // This method implements IFactory.Create
    X Create() {
        return new X();
    }
}
```

Java

Return type covariance is implemented in the Java programming language version J2SE 5.0. Parameter types have to be exactly the same (invariant) for method overriding, otherwise the method is overloaded with a parallel definition instead.

Generics were introduced in Java in Java 5.0 to allow type-safe generic programming. Unlike arrays, generic classes are neither covariant nor contravariant. For example, neither `List<String>` nor `List<Object>` is a subtype of the other:

```
// a is a single-element List of String
List<String> a = new ArrayList<String>();
a.add("foo");

// b is a List of Object
List<Object> b = a; // This is a compile-time error
```

However, generic type parameters can contain wildcards (a shortcut for an extra type parameter that is only used once). Example: Given a requirement for a method which operates on Lists, of any object, then the only operations that can be performed on the object are those for which the type relationships can be guaranteed to be safe.

```
// a is a single-element List of String
List<String> a = new ArrayList<String>();
a.add("foo");

// b is a List of anything
List<?> b = a;

// retrieve the first element
Object c = b.get(0);
// This is legal, because we can guarantee
// that the return type "?" is a subtype of Object

// Add an Integer to b.
b.add(new Integer (1));
// This is a compile-time error;
// we cannot guarantee that Integer is
// a subtype of the parameter type "?"
```

Wildcards can also be bound, e.g. "`? extends Foo`" or "`? super Foo`" for upper and lower bounds, respectively. This allows to refine permitted performance. Example: given a `List<? extends Foo>`, then an element can be retrieved and safely assigned to a `Foo` type (covariance). Given a `List<? super Foo>`, then a `Foo` object can be safely added as an element (contravariance).

Eiffel

Eiffel allows covariant return and parameter types in overriding methods. This is possible because Eiffel does not require subclasses to be substitutable for superclasses — that is, subclasses are not necessarily subtypes.

However, this can lead to surprises if subclasses with such covariant parameter types are operated upon presuming they were a more general class (polymorphism), leading to the possibility of compiler errors.

Nemerle

Nemerle supports declarations of covariance and contravariance (for interface, delegate and array types).

REALbasic

REALbasic added support for return type covariance in version 5.5. As with Java, the parameter types of the overriding method must be the same.

Scala

The Scala programming language originally supported use-site declarations of covariance and contravariance, but it turned out to be difficult to achieve consistency of usage-site type annotations, so that type errors were not uncommon. Because declaration-site

annotations provide excellent guidance on which methods should be generalized with lower bounds, Scala later switched from usage-site to declaration-site variance annotations. Though usage-site variance annotations allow a single class to have covariant as well as non-variant fragments, Scala's mixin composition makes it relatively easy to explicitly factor classes into covariant and non-variant fragments. Scala arrays are covariant in the base type, because they are implemented as functions yielding the base type.

Sather

Sather supports both covariance and contravariance. Calling convention for overridden methods are covariant with *out* arguments and return values, and contravariant with normal arguments (with the mode *in*).

Chapter 12

Liskov Substitution Principle and Effect System

Liskov substitution principle

Substitutability is a principle in object-oriented programming. It states that, if S is a subtype of T, then objects of type T in a computer program may be replaced with objects of type S (i.e., objects of type S may be *substituted* for objects of type T), without altering any of the desirable properties of that program (correctness, task performed, etc.). More formally, the **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called **(strong) behavioral subtyping**, that was initially introduced by Barbara Liskov in a 1987 conference keynote address entitled *Data abstraction and hierarchy*. It is a semantic rather than merely syntactic relation because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular. Liskov and Jeannette Wing formulated the principle succinctly in a 1994 paper as follows:

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

In the same paper, Liskov and Wing detailed their notion of behavioral subtyping in an extension of Hoare logic, which bears a certain resemblance with Bertrand Meyer's Design by Contract in that it considers the interaction of subtyping with pre- and postconditions.

The principle

Liskov's notion of a behavioral subtype defines a notion of substitutability for mutable objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).

Behavioral subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type. Behavioral subtyping is trivially undecidable in general: if *q* is the

property "method for always terminates", then it's impossible for a program (compiler) to verify that it holds true for some subtype *S* of *T* even if *q* does hold for *T*. The principle is useful however in reasoning about the design of class hierarchies.

Liskov's principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages (usually at the level of classes rather than types - see nominal vs. structural subtyping for the distinction):

- Contravariance of method arguments in the subtype.
- Covariance of return types in the subtype.
- No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

In addition to these, there is a number of behavioral conditions that subtype must meet. These are detailed in a terminology resembling that of design by contract methodology, leading to some restrictions on how contracts can interact with inheritance:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.
- History constraint (the "history rule"). Objects are regarded as being modifiable only through their methods (encapsulation). Since subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype. The history constraint prohibits this. It was the novel element introduced by Liskov and Wing. A violation of this constraint can be exemplified by defining a MutablePoint as a subtype of an ImmutablePoint. This is a violation of the history constraint, because in the history of the Immutable point, the state is always the same after creation, so it cannot include the history of a MutablePoint in general. Fields added to the subtype may however be safely modified because they are not observable through the supertype methods. One may derive a CircleWithFixedCenterButMutableRadius from ImmutablePoint without violating LSP.

Origins

The rules on pre- and postconditions are identical to those introduced by Bertrand Meyer in his 1988 book. Both Meyer, and later Pierre America, who was the first to use the term *behavioral subtyping*, gave proof-theoretic definitions of some behavioral subtyping notions, but their definitions did not take into account aliasing that may occur in programming language that supports references or pointers. Taking aliasing into account was the major improvement made by Liskov and Wing (1994), and a key ingredient is the history constraint. Under the definitions of Meyer and America a MutablePoint would be a behavioral subtype of ImmutablePoint, whereas LSP forbids this.

A typical violation

A typical example that violates LSP is a Square class that derives from a Rectangle class, assuming getter and setter methods exist for both width and height. The Square class always assumes that the width is equal with the height. If a Square object is used in a context where a Rectangle is expected, unexpected behavior may occur because the dimensions of a Square cannot (or rather should not) be modified independently. This problem cannot be easily fixed: if we can modify the setter methods in the Square class so that they preserve the Square invariant (i.e. keep the dimensions equal), then these methods will weaken (violate) the postconditions for the Rectangle setters, which state that dimensions can be modified independently. Violations of LSP, like this one, may or may not be a problem in practice, depending on the postconditions or invariants that are actually expected by the code that uses classes violating LSP. Mutability is a key issue here. If Square and Rectangle had only getter methods (i.e. they were immutable objects), then no violation of LSP could occur.

Effect system

An **effect system** is a formal system which describes the computational effects of computer programs, such as side effects. An effect system can be used to provide a compile-time checking of the possible effects of the program.

The effect system extends the notion of type to have an "effect" component, which comprises an **effect kind** and a **region**. The effect kind describes **what** is being done, and the region describes **with what** it is being done.

An effect system is typically an extension of a type system. The term "type and effect system" is sometimes used in this case. Often, a type of a value is denoted together with its effect as *type ! effect*, where both the type component and the effect component mention certain regions (for example, a type of a mutable memory cell is parameterized by the label of the memory region in which the cell resides).

Some examples of the behaviors that can be described by effect systems include:

- Reading, writing or allocating memory: the effect kind is *read*, *write*, *allocate* or *free*, and the region is the point of the program where allocation was performed (i.e., each program point where allocation is performed is assigned a unique label, and region information is statically propagated along the dataflow). Most functions working with memory will actually be polymorphic in the region variable: for example, a function that swaps two locations in memory will have type *forall r1 r2, unit ! {read r1, read r2, write r1, write r2}*.

- Working with resources, such as files: for example, the effect kind may be *open*, *read* and *close*, and again, the region is the point of the program where the resource is opened.
- Control transfers with continuations and long jumps: the effect kind may be *goto* (i.e. the piece of code may perform a jump) and *comefrom* (i.e. the piece of code may be the target of a jump), and the region denotes the point of the program from which or to which the jump may be performed.
- Java's checked exceptions are an example of an effect system: the effect kind is *throws* and the region is the type of the exception being thrown.

Effect systems may be used to prove the external purity of certain internally impure definitions: for example, if a function internally allocates and modifies a region of memory, but the function's type does not mention the region, then the corresponding effect may be erased from the function's effect.

Chapter 13

Nominative Type System and Structural Type System

Nominative type system

In computer science, a **nominal** or **nominative type system** (or name-based type system) is a major class of type system, in which compatibility and equivalence of data types is determined by explicit declarations and/or the name of the types. Nominative systems are used to determine if types are equivalent, as well as if a type is a subtype of another. It contrasts with structural systems, where comparisons are based on the structure of the types in question and do not require explicit declarations.

Nominal typing

Nominal typing means that two variables are type-compatible if and only if their declarations name the same type. For example, in C, two `struct` types with different names are never considered compatible, even if they have identical field declarations.

However, C also allows a `typedef` declaration, which introduces an alias for an existing type. These are merely syntactical and do not differentiate the type from its alias for the purpose of type checking. This feature, present in many languages, can result in a loss of type safety when (for example) the same primitive integer type is used in two semantically distinct ways. Haskell provides the C-style syntactic alias, as well as a declaration that does introduce a new, distinct type, isomorphic to an existing type.

Nominal subtyping

In a similar fashion, **nominal subtyping** means that one type is a subtype of another if and only if it is explicitly declared to be so in its definition. Nominally-typed languages typically enforce the requirement that declared subtypes be structurally compatible (though Eiffel allows non-compatible subtypes to be declared). However, subtypes which are structurally compatible "by accident", but not declared as subtypes, are not considered to be subtypes.

C, C++, C# and Java all primarily use both nominal typing and nominal subtyping.

Some nominally-subtyped languages, such as Java and C#, allow classes to be declared *final* (or *sealed* in C# terminology), indicating that no further subtyping is permitted.

Comparison

Nominal typing is useful at preventing accidental type equivalence, and is considered to have better type-safety than structural typing. The cost is a reduced flexibility, as, for example, nominal typing does not allow new super-types to be created without modification of the existing subtypes.

Structural type system

A **structural type system** (or property-based type system) is a major class of type system, in which type compatibility and equivalence are determined by the type's structure, and not by other characteristics such as its name or place of declaration. Structural systems are used to determine if types are equivalent and whether a type is a subtype of another. It contrasts with nominative systems, where comparisons are based on explicit declarations or the names of the types, and duck typing, in which only the part of the structure accessed at runtime is checked for compatibility.

In **structural typing**, an object or term is considered to be compatible with another type if for each feature within the second type, there must be a corresponding and identical feature in the first type. Some languages may differ on the details (such as whether the *features* must match in name). This definition is not symmetric, and includes subtype compatibility. Two such types are considered to be identical if each is compatible with the other.

ML, Objective Caml, and Go are examples of structurally-typed languages. C++ template functions exhibit structural typing on type arguments. HaXe uses structural typing, although classes are not structurally subtyped.

In languages which support subtype polymorphism, a similar dichotomy can be formed based on how the subtype relationship is defined. One type is a subtype of another if and only if it contains all the *features* of the base type (or subtypes thereof); the subtype may contain additional features (such as members not present in the base type, or stronger invariants).

Structural subtyping is arguably more flexible than nominative subtyping, as it permits the creation of *ad hoc* types and interfaces; in particular, it permits creation of a type which is a supertype of an existing type **T**, without modifying the definition of **T**. However this may not be desirable where the programmer wishes to create closed abstractions.

A pitfall of structural typing versus nominative typing is that two separately defined types intended for different purposes, each consisting of a pair of numbers, could be considered the same type by the type system, simply because they happen to have identical structure. One way this can be avoided is by creating one algebraic data type for one use of the pair and another algebraic data type for the other use.

Example

Objects in OCaml are structurally typed by the names and types of their methods.

Objects can be created directly ("immediate objects") without going through a nominative class. Classes only serve as functions for creating objects.

```
# let x =
    object
      val mutable x = 5
      method get_x = x
      method set_x y = x <- y
    end;;
val x : < get_x : int; set_x : int -> unit > = <obj>
```

Here the OCaml interactive runtime prints out the inferred type of the object for convenience. You can see that its type (`< get_x : int; set_x : int -> unit >`) is purely defined by its methods.

Let's define another object, which has the same methods and types of methods:

```
# let y =
    object
      method get_x = 2
      method set_x y = Printf.printf "%d\n" y
    end;;
val y : < get_x : int; set_x : int -> unit > = <obj>
```

You can see that OCaml considers them the same type. For example, the equality operator is typed to only take two values of the same type:

```
# x = y;;
- : bool = false
```

So they must be the same type, or else this wouldn't even type-check. This shows that equivalence of types is structural.

I can define a function that invokes a method:

```
# let set_to_10 a = a#set_x 10;;
val set_to_10 : < set_x : int -> 'a; .. > -> 'a = <fun>
```

The inferred type for the first argument (`< set_x : int -> 'a; .. >`) is interesting. The `..` means that the first argument can be any object which has a "set_x" method, which takes an int as argument.

So we can use it on object `x`:

```
# set_to_10 x;;
- : unit = ()
```

We can make another object that happens to have that method and method type; the other methods are irrelevant:

```
# let z =
    object
      method blahblah = 2.5
      method set_x y = Printf.printf "%d\n" y
    end;;
val z : < blahblah : float; set_x : int -> unit > = <obj>
```

The "set_to_10" function also works on it:

```
# set_to_10 z;;
10
- : unit = ()
```

This shows that compatibility for things like method invocation is determined by structure.

Let us define a type synonym for objects with only a "get_x" method and no other methods:

```
# type simpler_obj = < get_x : int >;;
type simpler_obj = < get_x : int >
```

Our object `x` is not of this type; but structurally, `x` is of a subtype of this type, since `x` contains a superset of its methods. So we can coerce `x` to this type:

```
# (x :> simpler_obj);;
- : simpler_obj = <obj>
# (x :> simpler_obj)#get_x;;
- : int = 10
```

But not object `z`, because it is not a structural subtype:

```
# (z :> simpler_obj);;
This expression cannot be coerced to type simpler_obj = < get_x : int
>;
it has type < blahblah : float; set_x : int -> unit > but is here used
with type
   < get_x : int; .. >
```

```
The first object type has no method get_x
```

This shows that compatibility for widening coercions are structural.

Chapter 14

Option Type and POPLmark Challenge

Option type

In programming languages (especially functional programming languages) and type theory, an **option type** or **maybe type** is a polymorphic type that represents encapsulation of an optional value; e.g. it is used as the return type of functions which may or may not return a meaningful value when they're applied. It consists of either an empty constructor (called *None* or *Nothing*), or a constructor encapsulating the original data type A (written *Just* A or *Some* A).

In the Haskell language, the option type (called *Maybe*) is defined as `data Maybe a = Just a | Nothing`. In the OCaml language, the option type is defined as `type 'a option = None | Some of 'a`.In the Scala language, Option is defined as parametrized abstract class `'.. Option[A] = if (x == null) None else Some(x)...` In the Standard ML language, the option type is defined as `datatype 'a option = NONE | SOME of 'a`.

In type theory, it may be written as: $A^? = A + 1$.

In languages that have discriminated unions (or sum types), as in most functional programming languages, option types can be expressed as the union of a unit type plus the encapsulated type.

In the Curry-Howard correspondence, option types are related to the annihilation law for $\vee$: x$\vee$1=1.

An option type can also be seen as a collection containing either a single element or zero elements.

The option monad

The option type is a monad under the following functions:

$$\text{return}\colon A \to A^? = a \mapsto \text{Just}\, a$$

$$\text{bind}\colon A^? \to (A \to B^?) \to B^? = a \mapsto f \mapsto \begin{cases} \text{Nothing} & \text{if } a = \text{Nothing} \\ f\, a' & \text{if } a = \text{Just}\, a' \end{cases}$$

We may also describe the option monad in terms of functions *return*, *fmap* and *join*, where the latter two are given by:

$$\text{fmap}\colon (A \to B) \to (A^? \to B^?) = f \mapsto a \mapsto \begin{cases} \text{Nothing} & \text{if } a = \text{Nothing} \\ \text{Just}\, f\, a' & \text{if } a = \text{Just}\, a' \end{cases}$$

$$\text{join}\colon A^{?^?} \to A^? = a \mapsto \begin{cases} \text{Nothing} & \text{if } a = \text{Nothing} \\ \text{Nothing} & \text{if } a = \text{Just Nothing} \\ \text{Just}\, a' & \text{if } a = \text{Just Just}\, a' \end{cases}$$

The option monad is an additive monad: it has *Nothing* as a zero constructor and the following function as a monadic sum:

$$\text{mplus}\colon A^? \to A^? \to A^? = a_1 \mapsto a_2 \mapsto \begin{cases} \text{Nothing} & \text{if } a_1 = \text{Nothing} \wedge a_2 = \text{Nothing} \\ \text{Just}\, a_2' & \text{if } a_1 = \text{Nothing} \wedge a_2 = \text{Just}\, a_2' \\ \text{Just}\, a_1' & \text{if } a_1 = \text{Just}\, a_1' \end{cases}$$

In fact, the resulting structure is an idempotent monoid.

POPLmark challenge

In type theory and programming languages, the **POPLmark challenge** is a set of benchmarks designed to evaluate the state of mechanization in the metatheory of programming languages, and to stimulate discussion and collaboration among a diverse cross section of the formal methods community. The challenge was initially proposed by the members of the *PL club* at the University of Pennsylvania, in association with collaborators around the world. The *Workshop on Mechanized Metatheory* is the main meeting of researchers participating in the challenge.

The design of the POPLmark benchmark is guided by features common to reasoning about programming languages. The challenge problems do not require the formalisation of large programming languages, but they do require sophistication in reasoning about:

Binding
: Most programming languages have some form of binding, ranging in complexity from the simple binders of simply typed lambda calculus to complex, potentially infinite binders needed in the treatment of record patterns.

Induction
: Properties such as subject reduction and strong normalisation often require complex induction arguments.

Reuse
: Furthering collaboration being a key aim of the challenge, the solutions are expected to contain reusable components that would allow researchers to share language features and designs without requiring them to start from scratch every time.

The problems

As of 2007, the POPLmark challenge is composed of three parts. Part 1 concerns solely the types of System $F_{<:}$ (System F with subtyping), and has problems such as:

1. Checking that the type system admits transitivity of subtyping.
2. Checking the transitivity of subtyping in the presence of records

Part 2 concerns the syntax and semantics of System $F_{<:}$. It concerns proofs of

1. Type safety for the pure fragment
2. Type safety in the presence of pattern matching

Part 3 concerns the usability of the formalisation of System $F_{<:}$. In particular, the challenge asks for:

1. Simulating and animating the operational semantics
2. Extracting useful algorithms from the formalisations

Several solutions have been proposed for parts of the POPLmark challenge, using tools such as Isabelle/HOL, Twelf, Coq and Matita.

Chapter 15

Recursive Data Type and Strong Typing

Recursive data type

In computer programming languages, a **recursive data type** (also known as a **recursively-defined**, **inductively-defined** or **inductive data type**) is a data type for values that may contain other values of the same type. Data of recursive types are usually viewed as directed graphs.

Sometimes the term "inductive data type" is used for algebraic data types which are not necessarily recursive.

Example

An example is the list type, in Haskell:

```
data List a = Nil | Cons a (List a)
```

This indicates that a list of a's is either an empty list or a **cons cell** containing an 'a' (the "head" of the list) and another list (the "tail").

Another example is a similar singly-linked type in Java:

```
class List<E> {
    E value;
    List<E> next;
}
```

This indicates that non-empty list of type E contains a data member of type E, and a reference to another List object for the rest of the list (or a null reference to indicate an empty rest of the list).

Theory

In type theory, a recursive type has the general form μα.T where the type variable α may appear in the type T and stands for the entire type itself.

For example, the natural number may be defined by the Haskell datatype:

```
data Nat = Zero | Succ Nat
```

In type theory, we would say: *nat* = μα.1 + α where the two arms of the sum type represent the Zero and Succ data constructors. Zero takes no arguments (thus represented by the unit type) and Succ takes another Nat (thus another element of μα.1 + α).

There are two forms of recursive types: the so-called isorecursive types, and equirecursive types. The two forms differ in how terms of a recursive type are introduced and eliminated.

Isorecursive types

With isorecursive types, the recursive type μα.*T* and its expansion (or *unrolling*) *T*[μα.*T* / α] are distinct (and disjoint) types with special term constructs, usually called *roll* and *unroll*, that form an isomorphism between them. To be precise: $roll : T[\mu\alpha.T/\alpha] \to \mu\alpha.T$ and $unroll : \mu\alpha.T \to T[\mu\alpha.T/\alpha]$, and these two are inverse functions.

In type synonyms

Recursion is not allowed in type synonyms in Miranda, OCaml (unless -rectypes flag is used), and Haskell; so for example the following Haskell types are illegal:

```
type Bad = (Int, Bad)
type Evil = Bool -> Evil
```

Instead, you must wrap it inside an algebraic data type (even if it only has one constructor):

```
data Good = Pair Int Good
data Fine = Fun (Bool->Fine)
```

This is because type synonyms, like typedefs in C, are replaced with their definition at compile time. (Type synonyms are not "real" types; they are just "aliases" for convenience of the programmer.) But if you try to do this with a recursive type, it will loop infinitely because no matter how many times you substitute it, it still refers to itself. (i.e. "Bad" will grow indefinitely: (Int, (Int, (Int, ... Bad))...))

Another way to see it is that a level of indirection (the algebraic data type) is required to allow the isorecursive type system to figure out when to *roll* and *unroll*.

Equirecursive types

Under equirecursive rules, a recursive type $\mu\alpha.T$ and its unrolling $T[\mu\alpha.T / \alpha]$ are *equal* -- that is, those two type expressions are understood to denote the same type. In fact, most theories of equirecursive types go further and essentially stipulate that any two type expressions with the same "infinite expansion" are equivalent. As a result of these rules, equirecursive types contribute significantly more complexity to a type system than isorecursive types do. Algorithmic problems such as type checking and type inference are more difficult for equirecursive types as well. Since direct comparison does not make sense on an equirecursive type, they can be converted into a canonical form in O(n log n) time, which can easily be compared.

Equirecursive types capture the form of self-referential (or mutually referential) type definitions seen in procedural and object-oriented programming languages, and also arise in type-theoretic semantics of objects and classes. In functional programming languages, isorecursive types (in the guise of datatypes) are far more ubiquitous.

Strong typing

In computer science and computer programming, a type system is said to feature **strong typing** when it specifies one or more restrictions on how operations involving values of different data types can be intermixed. The opposite of strong typing is *weak typing*.

Interpretation

Most generally, "strong typing" implies that the programming language places severe restrictions on the intermixing that is permitted to occur, preventing the compiling or running of source code which uses data in what is considered to be an invalid way. For instance, an integer division operation may not be used upon strings; a procedure which operates upon linked lists may not be used upon numbers. However, the nature and strength of these restrictions is highly variable.

Benjamin C. Pierce, author of *Types and Programming Languages* and *Advanced Types and Programming Languages*, says, "I spent a few weeks... trying to sort out the terminology of "strongly typed," "statically typed," "safe," etc., and found it amazingly difficult.... The usage of these terms is so various as to render them almost useless." Luca Cardelli's article Typeful Programming describes strong typing simply as the absence of unchecked run-time type errors. In other writing, the absence of unchecked run-time errors is referred to as *safety* or *type safety*; Tony Hoare's early papers call this property *security*.

Example

	Weak Typing	Strong Typing

Pseudocode	`a = 2`
	`b = '2'`
	
	`concatenate(a, b)  # Returns '22'`
	`add(a, b)  # Returns 4`
		`a = 2`
		`b = '2'`
		
		`#concatenate(a, b)  # Type Error`
		`#add(a, b)  # Type Error`
		`concatenate(str(a), b)  # Returns '22'`
		`add(a, int(b))  # Returns 4`

Languages	Perl, Rexx	Java, Python

Meanings in computer literature

Some of the factors which writers have qualified as "strong typing" include:

- **Strong guarantees about the run-time behavior of a program** before program execution, whether provided by static analysis, the execution semantics of the language or another mechanism.
- **Type safety**; that is, at compile or run time, the rejection of operations or function calls which attempt to disregard data types. In a more rigorous setting, type safety is proved about a formal language by proving progress and preservation.
- The guarantee that **a well-defined error or exceptional behavior** (as opposed to an undefined behavior) **occurs as soon as a type-matching failure happens** at runtime, or, as a special case of that with even stronger constraints, the guarantee that type-matching failures would *never* happen at runtime (which would also satisfy the constraint of "no undefined behavior" after type-matching failures, since the latter would never happen anyway).
- The mandatory requirement, by a language definition, of **compile-time checks** for type constraint violations. That is, the compiler ensures that operations only occur on operand types that are valid for the operation.
- **Fixed and invariable typing of data objects**. The type of a given data object does not vary over that object's lifetime. For example, class instances may not have their class altered.
- The **absence of ways to evade the type system**. Such evasions are possible in languages that allow programmer access to the underlying representation of values, i.e., their bit-pattern.
- **Omission of implicit type conversion**, that is, conversions that are inserted by the compiler on the programmer's behalf. For these authors, a programming language is strongly typed if type conversions are allowed only when an explicit notation, often called a *cast*, is used to indicate the desire of converting one type to another.
- **Disallowing** any kind of **type conversion**. Values of one type cannot be converted to another type, explicitly or implicitly.

- A **complex, fine-grained type system** with compound types.

Variation across programming languages

Note that some of these definitions are contradictory, others are merely orthogonal, and still others are special cases (with additional constraints) of other, more "liberal" (less strong) definitions. Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed. For instance:

- Java, C#, Pascal, Ada and C require all variables to have a defined type and support the use of explicit casts of arithmetic values to other arithmetic types. Java, C#, Ada and Pascal are often said to be more strongly typed than C, a claim that is probably based on the fact that C supports more kinds of implicit conversions, and C also allows pointer values to be explicitly cast while Java and Pascal do not. Java itself may be considered more strongly typed than Pascal as manners of evading the static type system in Java are controlled by the Java Virtual Machine's dynamic type system. C# is similar to Java in that respect, though it allows to disable dynamic type checking by explicitly putting code segments in an "unsafe context".
- The object-oriented programming languages Smalltalk, Ruby, Python, and Self are all "strongly typed" in the sense that typing errors are prevented at runtime and they do little implicit type conversion, but these languages make no use of static type checking: the compiler does not check or enforce type constraint rules. The term duck typing is now used to describe the dynamic typing paradigm used by the languages in this group.
- The Lisp family of languages are all "strongly typed" in the sense that typing errors are prevented at runtime. Some Lisp dialects like Common Lisp do support various forms of type declarations and some compilers (CMUCL and related) are using these declarations together with type inference to enable various optimizations and also limited forms of compile time type checks.
- Standard ML, Objective Caml and Haskell have purely static type systems, in which the compiler automatically infers a precise type for all values. These languages (along with most functional languages) are considered to have stronger type systems than Java, as they permit no implicit type conversions. While OCaml's libraries allow one form of evasion (*Object magic*), this feature remains unused in most applications.
- Visual Basic is a hybrid language. In addition to including statically typed variables, it includes a "Variant" data type that can store data of any type. Its implicit casts are fairly liberal where, for example, one can sum string variants and pass the result into an integer literal.
- Assembly language and Forth have been said to be *untyped*. There is no type checking; it is up to the programmer to ensure that data given to functions is of the appropriate type. Any type conversion required is explicit.

For this reason, writers who wish to write unambiguously about type systems often eschew the term "strong typing" in favor of specific expressions such as "static typing" or "type safety".