Logic Programming

Frank Pfenning
Carnegie Mellon University

Draft of January 2, 2007

Notes for a course given at Carnegie Mellon University, Fall 2006. Materials available at `http://www.cs.cmu.edu/~fp/courses/lp`. Please send comments to `fp@cs.cmu.edu`.

15-819K: Logic Programming

Lecture 1

Logic Programming

Frank Pfenning

August 29, 2006

In this first lecture we give a brief introduction to logic programming. We also discuss administrative details of the course, although these are not included here, but can be found on the course web page.[1]

1.1 Computation vs. Deduction

Logic programming is a particular way to approach programming. Other paradigms we might compare it to are imperative programming or functional programming. The divisions are not always clear-cut—a functional language may have imperative aspects, for example—but the mindset of various paradigms is quite different and determines how we design and reason about programs.

To understand logic programming, we first examine the difference between computation and deduction. To *compute* we start from a given expression and, according to a fixed set of rules (the program) generatee a result. For example, $15 + 26 \rightarrow (1 + 2 + 1)1 \rightarrow (3 + 1)1 \rightarrow 41$. To *deduce* we start from a conjecture and, according to a fixed set of rules (the axioms and inference rules), try to construct a proof of the conjecture. So computation is mechanical and requires no ingenuity, while deduction is a creative process. For example, $a^n + b^n \neq c^n$ for $n > 2$, *. . . 357 years of hard work . . .*, QED.

Philosophers, mathematicians, and computer scientists have tried to unify the two, or at least to understand the relationship between them for centuries. For example, George Boole[2] succeeded in reducing a certain class

[1] `http://www.cs.cmu.edu/~fp/courses/lp/`

[2] 1815–1864

of logical reasoning to computation in so-called Boolean algebras. Since the fundamental undecidability results of the 20th centuries we know that not everything we can reason about is in fact mechanically computable, even if we follow a well-defined set of formal rules.

In this course we are interested in a connection of a different kind. A first observation is that computation can be seen as a limited form of deduction because it establishes theorems. For example, $15 + 26 = 41$ is both the result of a computation, and a theorem of arithmetic. Conversely, deduction can be considered a form of computation if we fix a strategy for proof search, removing the guesswork (and the possibility of employing ingenuity) from the deductive process.

This latter idea is the foundation of logic programming. Logic program computation proceeds by proof search according to a fixed strategy. By knowing what this strategy is, we can implement particular algorithms in logic, and execute the algorithms by proof search.

1.2 Judgments and Proofs

Since logic programming computation is proof search, to study logic programming means to study proofs. We adopt here the approach by Martin-Löf [3]. Although he studied logic as a basis for functional programming rather than logic programming, his ideas are more fundamental and therefore equally applicable in both paradigms.

The most basic notion is that of a *judgment*, which is an object of knowledge. We know a judgment because we have evidence for it. The kind of evidence we are most interested in is a *proof*, which we display as a *deduction* using *inference rules* in the form

$$\frac{J_1 \ldots J_n}{J}\ R$$

where R is the name of the rule (often omitted), J is the judgment established by the inference (the *conclusion*), and $J_1, \ldots, J_n$ are the *premisses* of the rule. We can read it as

If J_1 and $\cdots$ and J_n then we can conclude J by virtue of rule R.

By far the most common judgment is the truth of a proposition A, which we write as $A\ true$. Because we will be occupied almost exclusively with the thruth of propositions for quite some time in this course we generally omit the trailing "$true$". Other examples of judgments on propositions are

A *false* (A is false), A *true at* t (A is true at time t, the subject of temporal logic), or K *knows* A (K knows that A is true, the subject of epistemic logic).

To give some simple examples we need a language to express propositions. We start with *terms* t that have the form $f(t_1, \ldots, t_n)$ where f is a *function symbol* of arity n and $t_1, \ldots, t_n$ are the arguments. Terms can have variables in them, which we generally denote by upper-case letters. *Atomic propositions* P have the form $p(t_1, \ldots, t_n)$ where p is a *predicate symbol* of arity n and $t_1, \ldots, t_n$ are its arguments. Later we will introduce more general forms of propositions, built up by logical connectives and quantifiers from atomic propositions.

In our first set of examples we represent natural numbers $0, 1, 2, \ldots$ as terms of the form $\mathsf{z}, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), \ldots$, using two function symbols (z of arity 0 and s of arity 1).[3] The first predicate we consider is even of arity 1. Its meaning is defined by two inference rules:

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz} \qquad\qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evs}$$

The first rule, evz, expresses that 0 is even. It has no premiss and therefore is like an axiom. The second rule, evs, expresses that if N is even, then $\mathsf{s}(\mathsf{s}(N))$ is also even. Here, N is a *schematic variable* of the inference rule: every instance of the rule where N is replaced by a concrete term represents a valid inference. We have no more rules, so we think of these two as defining the predicate even completely.

The following is a trivial example of a deduction, showing that 4 is even:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{evs}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))}\ \mathsf{evs}$$

Here, we used the rule evs twice: once with $N = \mathsf{z}$ and once with $N = \mathsf{s}(\mathsf{s}(\mathsf{z}))$.

1.3 Proof Search

To make the transition from inference rules to logic programming we need to impose a particular strategy. Two fundamental ideas suggest themselves: we could either search backward from the conjecture, growing a

[3] This is not how numbers are represented in practical logic programming languages such as Prolog, but it is a convenient source of examples.

(potential) proof tree upwards, or we could work forwards from the axioms applying rules until we arrive at the conjecture. We call the first one *goal-directed* and the second one *forward-reasoning*. In the logic programming literature we find the terminology *top-down* for goal-directed, and *bottom-up* for forward-reasoning, but this goes counter to the direction in which the proof tree is constructed. Logic programming was conceived with goal-directed search, and this is still the dominant direction since it underlies Prolog, the most popular logic programming language. Later in the class, we will also have an opportunity to consider forward reasoning.

In the first approximation, the goal-directed strategy we apply is very simple: given a conjecture (called the *goal*) we determine which inference rules might have been applied to arrive at this conclusion. We select one of them and then recursively apply our strategy to all the premisses as subgoals. If there are no premisses we have completed the proof of the goal. We will consider many refinements and more precise descriptions of search in this course.

For example, consider the conjecture $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$. We now execute the logic program consisting of the two rules evz and evs to either prove or refute this goal. We notice that the only rule with a matching conclusion is evs. Our partial proof now looks like

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{even}(\mathsf{z})\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{evs}$$

with $\mathsf{even}(\mathsf{z})$ as the only subgoal.

Considering the subgoal $\mathsf{even}(\mathsf{z})$ we see that this time only the rule evz could have this conclusion. Moreover, this rule has no premisses so the computation terminates successfully, having found the proof

$$\dfrac{\dfrac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{evs}.$$

Actually, most logic programming languages will not show the proof in this situation, but only answer `yes` if a proof has been found, which means the conjecture was true.

Now consider the goal $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))$. Clearly, since 3 is not even, the computation must fail to produce a proof. Following our strategy, we first

reduce this goal using the evs rule to the subgoal $\mathsf{even}(\mathsf{s}(\mathsf{z}))$, with the incomplete proof

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{even}(\mathsf{s}(\mathsf{z}))\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))}\ \mathsf{evs}.$$

At this point we note that there is no rule whose conclusion matches the goal $\mathsf{even}(\mathsf{s}(\mathsf{z}))$. We say proof search *fails*, which will be reported back as the result of the computation, usually by printing `no`.

Since we think of the two rules as the complete definition of even we conclude that $\mathsf{even}(\mathsf{s}(\mathsf{z}))$ is *false*. This example illustrates *negation as failure*, which is a common technique in logic programming. Notice, however, that there is an asymmetry: in the case where the conjecture was true, search constructed an explicit proof which provides evidence for its truth. In the case where the conjecture was false, no evidence for its falsehood is immediately available. This means that negation does not have first-class status in logic programming.

1.4 Answer Substitutions

In the first example the response to a goal is either `yes`, in which case a proof has been found, or `no`, if all attempts at finding a proof fail finitely. It is also possible that proof search does not terminate. But how can we write logic programs to compute values?

As an example we consider programs to compute sums and differences of natural numbers in the representation from the previous section. We start by specifying the underlying *relation* and then illustrate how it can be used for computation. The relation in this case is $\mathsf{plus}(m, n, p)$ which should hold if $m + n = p$. We use the recurrence

$$\begin{array}{lcl}(m+1)+n & = & (m+n)+1\\ 0+n & = & n\end{array}$$

as our guide because it counts down the first argument to 0. We obtain

$$\dfrac{\mathsf{plus}(M,N,P)}{\mathsf{plus}(\mathsf{s}(M),N,\mathsf{s}(P))}\ \mathsf{ps} \qquad\qquad \dfrac{}{\mathsf{plus}(\mathsf{z},N,N)}\ \mathsf{pz}.$$

Now consider a goal of the form $\mathsf{plus}(\mathsf{s}(\mathsf{z}),\mathsf{s}(\mathsf{z}),R)$ where R is an unknown. This represents the question if there exists an R such that the relation $\mathsf{plus}(\mathsf{s}(\mathsf{z}),\mathsf{s}(\mathsf{z}),R)$ holds. Search not only constructs a proof, but also a term t for R such that $\mathsf{plus}(\mathsf{s}(\mathsf{z}),\mathsf{s}(\mathsf{z}),t)$ is true.

For the original goal, $\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), R)$, only the rule ps could apply because of a mismatch between z and $\mathsf{s}(\mathsf{z})$ in the first argument to plus in the conclusion. We also see that the R must have the form $\mathsf{s}(P)$ for some P, although we do not know yet what P should be.

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P)\end{array}}{\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)$$

For the subgoal only the pz rule applies and we see that P must equal $\mathsf{s}(\mathsf{z})$.

$$\dfrac{\dfrac{}{\mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P)}\ \mathsf{pz} \quad \text{with } P = \mathsf{s}(\mathsf{z})}{\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)$$

If we carry out the substitutions we obtain the complete proof

$$\dfrac{\dfrac{}{\mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}))}\ \mathsf{pz}}{\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{ps}$$

which is explicit evidence that $1 + 1 = 2$. Instead of the full proof, implementations of logic programming languages mostly just print the substitution for the unknowns in the original goal, in this case $R = \mathsf{s}(\mathsf{s}(\mathsf{z}))$.

Some terminology of logic programming: the original goal is called the *query*, its unknowns are *logic variables*, and the result of the computation is an *answer substitution* for the logic variables, suppressing the proof.

1.5 Backtracking

Sometimes during proof search the goal matches the conclusion of more than one rule. This is called a *choice point*. When we reach a choice point we pick the first among the rules that match, in the order they were presented. If that attempt at a proof fails, we try the second one that matches, and so on. This process is called *backtracking*.

As an example, consider the query $\mathsf{plus}(M, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))$, intended to compute an m such that $m + 1 = 2$, that is, $m = 2 - 1$. This demonstrates that we can use the same logic program (here: the definition of the plus predicate) in different ways (before: addition, now: subtraction).

The conclusion of the rule pz, $\mathsf{plus}(\mathsf{z}, N, N)$, does not match because the second and third argument of the query are different. However, the rule ps applies and we obtain

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_1, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}))\end{array}}{\mathsf{plus}(M, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point both rules, ps and pz, match. Because the rule ps is listed first, leading to

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_2, \mathsf{s}(\mathsf{z}), \mathsf{z})\end{array}}{\mathsf{plus}(M_1, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}))}\ \mathsf{ps} \quad \text{with } M_1 = \mathsf{s}(M_2)}{\mathsf{plus}(M, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point no rule applies at all and this attempt fails. So we return to our earlier choice point and try the second alternative, pz.

$$\dfrac{\dfrac{}{\mathsf{plus}(M_1, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}))}\ \mathsf{pz} \quad \text{with } M_1 = \mathsf{z}}{\mathsf{plus}(M, \mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point the proof is complete, with the answer substitution $M = \mathsf{s}(\mathsf{z})$.

Note that with even a tiny bit of foresight we could have avoided the failed attempt by picking the rule pz first. But even this small amount of ingenuity cannot be permitted: in order to have a satisfactory programming language we must follow every step prescribed by the search strategy.

1.6 Subgoal Order

Another kind of choice arises when an inference rule has multiple premises, namely the order in which we try to find a proof for them. Of course, logically the order should not be relevant, but operationally the behavior of a program can be quite different.

As an example, we define of $\mathsf{times}(m, n, p)$ which should hold if $m \times n = p$. We implement the recurrence

$$\begin{array}{lcl} 0 \times n & = & 0 \\ (m+1) \times n & = & (m \times n) + n \end{array}$$

in the form of the following two inference rules.

$$\frac{}{\mathsf{times}(\mathsf{z}, N, \mathsf{z})}\ \mathsf{tz} \qquad \frac{\mathsf{times}(M, N, P) \quad \mathsf{plus}(P, N, Q)}{\mathsf{times}(\mathsf{s}(M), N, Q)}\ \mathsf{ts}$$

As an example we compute $1 \times 2 = Q$. The first step is determined.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \mathsf{times}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})), P) & \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)\end{array}}{\mathsf{times}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{ts}$$

Now if we solve the left subgoal first, there is only one applicable rule which forces $P = \mathsf{z}$

$$\frac{\begin{array}{cc}\dfrac{}{\mathsf{times}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})), P)}\ \mathsf{ts}\ (P = \mathsf{z}) & \vdots \\ & \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)\end{array}}{\mathsf{times}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{ts}$$

Now since $P = \mathsf{z}$, there is only one rule that applies to the second subgoal and we obtain correctly

$$\frac{\dfrac{}{\mathsf{times}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})), P)}\ \mathsf{ts}\ (P = \mathsf{z}) \quad \dfrac{}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{pz}\ (Q = \mathsf{s}(\mathsf{s}(\mathsf{z})))}{\mathsf{times}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{ts}.$$

On the other hand, if we solve the right subgoal $\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)$ first we have no information on P and Q, so both rules for plus apply. Since ps is given first, the strategy discussed in the previous section means that we try it first, which leads to

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \mathsf{times}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})), P) & \dfrac{\mathsf{plus}(P_1, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q_1)}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{ps}\ (P = \mathsf{s}(P_1), Q = \mathsf{s}(Q_1))\end{array}}{\mathsf{times}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), Q)}\ \mathsf{ts}.$$

Again, rules ps and ts are both applicable, with ps listed further, so we

continue:

$$\dfrac{\dfrac{\vdots}{\mathsf{times}(\mathsf{z},\mathsf{s}(\mathsf{s}(\mathsf{z})),P)} \quad \dfrac{\dfrac{\dfrac{\vdots}{\mathsf{plus}(P_2,\mathsf{s}(\mathsf{s}(\mathsf{z})),Q_2)}}{\mathsf{plus}(P_1,\mathsf{s}(\mathsf{s}(\mathsf{z})),Q_1)}\ \mathsf{ps}\ (P_1=\mathsf{s}(P_2),Q_1=\mathsf{s}(Q_2))}{\mathsf{plus}(P,\mathsf{s}(\mathsf{s}(\mathsf{z})),Q)}\ \mathsf{ps}\ (P=\mathsf{s}(P_1),Q=\mathsf{s}(Q_1))}{\mathsf{times}(\mathsf{s}(\mathsf{z}),\mathsf{s}(\mathsf{s}(\mathsf{z})),Q)}\ \mathsf{ts}$$

It is easy to see that this will go on indefinitely, and computation will not terminate.

This examples illustrate that the order in which subgoals are solved can have a strong impact on the computation. Here, proof search either completes in two steps or does not terminate. This is a consequence of fixing an operational reading for the rules. The standard solution is to attack the subgoals in left-to-right order. We observe here a common phenomenon of logic programming: two definitions, entirely equivalent from the logical point of view, can be very different operationally. Actually, this is also true for functional programming: two implementations of the same function can have very different complexity. This debunks the myth of "declarative programming"—the idea that we only need to specify the problem rather than design and implement an algorithm for its solution. However, we can assert that both specification and implementation can be expressed in the language of logic. As we will see later when we come to logical frameworks, we can integrate even correctness proofs into the same formalism!

1.7 Prolog Notation

By far the most widely used logic programming language is Prolog, which actually is a family of closely related languages. There are several good textbooks, language manuals, and language implementations, both free and commercial. A good resource is the FAQ[4] of the Prolog newsgroup[5] For this course we use GNU Prolog[6] although the programs should run in just about any Prolog since we avoid the more advanced features.

The two-dimensional presentation of inference rules does not lend itself

[4] `http://www.cs.kuleuven.ac.be/~remko/prolog/faq/files/faq.html`
[5] `news://comp.lang.prolog/`
[6] `http://gnu-prolog.inria.fr/`

to a textual format. The Prolog notation for a rule

$$\frac{J_1 \ldots J_n}{J}\ R$$

is

$$J \leftarrow J_1, \ldots, J_n.$$

where the name of the rule is omitted and the left-pointing arrow is rendered as ':-' in a plain text file. We read this as

$$J \textit{ if } J_1 \textit{ and } \cdots \textit{ and } J_n.$$

Prolog terminology for an inference rule is a *clause*, where J is the *head* of the clause and $J_1, \ldots, J_n$ is the body. Therefore, instead of saying that we "*search for an inference rule whose conclusion matches the conjecture*", we say that we "*search for a clause whose head matches the goal*".

As an example, we show the earlier programs in Prolog notation.

```
even(z).
even(s(s(N))) :- even(N).

plus(s(M), N, s(P)) :- plus(M, N, P).
plus(z, N, N).

times(z, N, z).
times(s(M), N, Q) :-
        times(M, N, P),
        plus(P, N, Q).
```

Clauses are tried in the order they are presented in the program. Subgoals are solved in the order they are presented in the body of a clause.

1.8 Unification

One important operation during search is to determine if the conjecture matches the conclusion of an inference rule (or, in logic programming terminology, if the goal unifies with the head of a clause). This operation is a bit subtle, because the the rule may contain schematic variables, and the the goal may also contain variables.

As a simple example (which we glossed over before), consider the goal $\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), R)$ and the clause $\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P)$. We

need to find some way to instantiate M, N, and P in the clause head and R in the goal such that $\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), R) = \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))$.

Without formally describing an algorithm yet, the intuitive idea is to match up corresponding subterms. If one of them is a variable, we set it to the other term. Here we set $M = \mathsf{z}$, $N = \mathsf{s}(\mathsf{z})$, and $R = \mathsf{s}(P)$. P is arbitrary and remains a variable. Applying these equations to the body of the clause we obtain $\mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P)$ which will be the subgoal with another logic variable, P.

In order to use the other clause for plus to solve this goal we have to solve $\mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P) = \mathsf{plus}(\mathsf{z}, N, N)$ which sets $N = \mathsf{s}(\mathsf{z})$ and $P = \mathsf{s}(\mathsf{z})$.

This process is called *unification*, and the equations for the variables we generate represent the *unifier*. There are some subtle issues in unification. One is that the variables in the clause (which really are schematic variables in an inference rule) should be renamed to become fresh variables each time a clause is used so that the different instances of a rule are not confused with each other. Another issue is exemplified by the equation $N = \mathsf{s}(\mathsf{s}(N))$ which does not have a solution: the right-hand side will have have two more successors than the left-hand side so the two terms can never be equal. Unfortunately, Prolog does not properly account for this and treats such equations incorrectly by building a circular term (which is definitely not a part of the underlying logical foundation). This could come up if we pose the query $\mathsf{plus}(\mathsf{z}, N, \mathsf{s}(N))$: "*Is there an n such that $0 + n = n + 1$.*"

We discuss the reasons for Prolog's behavior later in this course (which is related to efficiency), although we do not subscribe to it because it subverts the logical meaning of programs.

1.9 Beyond Prolog

Since logic programming rests on an operational interpretation of logic, we must study various logics as well as properties of proof search in these logics in order to understand logic programming. We will therefore spend a fair amount of time in this course isolating logical principles. Only in this way can we push the paradigm to its limits without departing too far from what makes it beautiful: its elegant logical foundation.

Roughly, we repeat the following steps multiple times in the course, culminating in an incredibly rich language that can express backtracking search, concurrency, saturation, and even correctness proofs for many programs in a harmonious whole.

1. Design a logic in a foundationally and philosophically sound manner.

2. Isolate a fragment of the logic based on proof search criteria.
3. Give an informal description of its operational behavior.
4. Explore programming techniques and idioms.
5. Formalize the operational semantics.
6. Implement a high-level interpreter.
7. Study properties of the language as a whole.
8. Develop techniques for reasoning about individual programs.
9. Identify limitations and consider how they might be addressed, either by logical or operational means.
10. Go to step 1.

Some of the logics we will definitely consider are intuitionistic logic, modal logic, higher-order logic, and linear logic, and possibly also temporal and epistemic logics. Ironically, even though logic programming derives from logic, the language we have considered so far (which is the basis of Prolog) does not require any logical connectives at all, just the mechanisms of judgments and inference rules.

1.10 Historical Notes

Logic programming and the Prolog language are credited to Alain Colmerauer and Robert Kowalski in the early 1970s. Colmerauer had been working on a specialized theorem prover for natural language processing, which eventually evolved to a general purpose language called Prolog (for *Programmation en Logique*) that embodies the operational reading of clauses formulated by Kowalski. Interesting accounts of the birth of logic programming can be found in papers by the Colmerauer and Roussel [1] and Kowalski [2].

We like Sterling and Shapiro's *The Art of Prolog* [4] as a good introductory textbook for those who already know how to program and we recommends O'Keefe's *The Craft of Prolog* as a second book for those aspiring to become real Prolog hackers. Both of these are somewhat dated and do not cover many modern developments, which are the focus of this course. We therefore do not use them as textbooks here.

1.11 Exercises

Exercise 1.1 *A different recurrence for addition is*

$$\begin{array}{lcl}(m+1)+n & = & m+(n+1)\\ 0+n & = & n\end{array}$$

Write logic programs for addition on numbers in unary notation based on this recurrence. What kind of query do we need to pose to compute differences correctly?

Exercise 1.2 *Determine if the* times *predicate can be used to calculate exact division, that is, given m and n find q such that $m = n \times q$ and fail if no such q exists. If not, give counterexamples for different ways that* times *could be invoked and write another program* divexact *to perform exact divison. Also write a program to calculate both quotient and remainder of two numbers.*

Exercise 1.3 *We saw that the* plus *predicate can be used to compute sums and differences. Find other creative uses for this predicate without writing any additional code.*

Exercise 1.4 *Devise a representation of natural numbers in binary form as terms. Write logic programs to add and multiply binary numbers, and to translate between unary and binary numbers. Can you write a single relation that can be executed to translate in both directions?*

1.12 References

[1] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *Conference on the History of Programming Languages (HOPL-II), Preprints*, pages 37–52, Cambridge, Massachusetts, April 1993.

[2] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[3] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[4] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2nd edition edition, 1994.

15-819K: Logic Programming

Lecture 2

Data Structures

Frank Pfenning

August 31, 2006

In this second lecture we introduce some simple data structures such as lists, and simple algorithms on them such as as quicksort or mergesort. We also introduce some first considerations of types and modes for logic programs.

2.1 Lists

Lists are defined by two constructors: the empty list nil and the constructor cons which takes an element and a list, generating another list. For example, the list a, b, c would be represented as $\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil})))$. The official Prolog notation for nil is `[]`, and for $\mathsf{cons}(h, t)$ is `.(`h`, `t`)`, overloading the meaning of the period '`.`' as a terminator for clauses and a binary function symbol. In practice, however, this notation for cons is rarely used. Instead, most Prolog programs use `[`h`|`t`]` for $\mathsf{cons}(h, t)$.

There is also a sequence notation for lists, so that a, b, c can be written as `[a, b, c]`. It could also be written as `[a | [b | [c | []]]]` or `[a, b | [c, []]]`. Note that all of these notations will be parsed into the same internal form, using nil and cons. We generally follow Prolog list notation in these notes.

2.2 Type Predicates

We now return to the definition of plus from the previous lecture, except that we have reversed the order of the two clauses.

```
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

In view of the new list constructors for terms, the first clause now looks wrong. For example, with this clause we can prove

```
plus(s(z), [a, b, c], s([a, b, c])).
```

This is absurd: what does it mean to add 1 and a list? What does the term `s([a, b, c])` denote? It is clearly neither a list nor a number.

From the modern programming language perspective the answer is clear: the definition above lacks *types*. Unfortunately, Prolog (and traditional predicate calculus from which it was originally derived) do not distinguish terms of different types. The historical answer for why these languages have just a single type of terms is that types can be defined as unary predicates. While this is true, it does not account for the pragmatic advantage of distinguishing meaningful propositions from those that are not. To illustrate this, the standard means to correct the example above would be to define a predicate nat with the rules

$$\frac{}{\mathsf{nat}(\mathsf{z})}\ \mathsf{nz} \qquad\qquad \frac{\mathsf{nat}(N)}{\mathsf{nat}(\mathsf{s}(N))}\ \mathsf{ns}$$

and modify the base case of the rules for addition

$$\frac{\mathsf{nat}(N)}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad\qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

One of the problems is that now, for example, $\mathsf{plus}(\mathsf{z}, \mathsf{nil}, \mathsf{nil})$ is *false*, when it should actually be meaningless. Many problems in debugging Prolog programs can be traced to the fact that propositions that should be meaningless will be interpreted as either true or false instead, incorrectly succeeding or failing. If we transliterate the above into Prolog, we get:

```
nat(z).
nat(s(N)) :- nat(N).

plus(z, N, N) :- nat(N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

No self-respecting Prolog programmer would write the `plus` predicate this way. Instead, he or she would omit the type test in the first clause leading to the earlier program. The main difference between the two is whether meaningless clauses are false (with the type test) or true (without the type test). One should then annotate the predicate with the intended domain.

```
% plus(m, n, p) iff m + n = p for nat numbers m, n, p.
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

It would be much preferable from the programmer's standpoint if this informal comment were a formal type declaration, and an illegal invocation of `plus` were a compile-time error rather than leading to silent success or failure. There has been some significant research on types systems and type checking for logic programming languages [5] and we will talk about types more later in this course.

2.3 List Types

We begin with the type predicates defining lists.

```
list([]).
list([X|Xs]) :- list(Xs).
```

Unlike languages such as ML, there is no test whether the elements of a list all have the same type. We could easily test whether something is a list of natural numbers.

```
natlist([]).
natlist([N|Ns]) :- nat(N), natlist(Ns).
```

The generic test, whether we are presented with a homogeneous list, all of whose elements satisfy some predicate `P`, would be written as:

```
plist(P, []).
plist(P, [X|Xs]) :- P(X), plist(P, Xs).
```

While this is legal in some Prolog implementations, it can not be justified from the underlying logical foundation, because `P` stands for a predicate and is an argument to another predicate, `plist`. This is the realm of higher-order logic, and a proper study of it requires a development of *higher-order logic programming* [3, 4]. In Prolog the goal `P(X)` is a *meta-call*, often written as `call(P(X))`. We will avoid its use, unless we develop higher-order logic programming later in this course.

2.4 List Membership and Disequality

As a second example, we consider membership of an element in a list.

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

In Prolog syntax:

```
% member(X, Ys) iff X is a member of list Ys
member(X, [X|Ys]).
member(X, [Y|Ys]) :- member(X, Ys).
```

Note that in the first clause we have omitted the check whether `Ys` is a proper list, making it part of the presupposition that the second argument to `member` is a list.

Already, this very simple predicate has some subtle points. To show the examples, we use the Prolog notation `?-` A`.` for a query A. After presenting the first answer substitution, Prolog interpreters issue a prompt to see if another solution is desired. If the user types ';' the interpreter will backtrack to see if another solution can be found. For example, the query

```
?- member(X, [a,b,a,c]).
```

has four solutions, in the order

```
X = a;
X = b;
X = a;
X = c.
```

Perhaps surprisingly, the query

```
?- member(a, [a,b,a,c]).
```

succeeds twice (both with the empty substitution), once for the first occurrence of `a` and once for the second occurrence.

If `member` is part of a larger program, backtracking of a later goal could lead to unexpected surprises when `member` succeeds again. There could also be an efficiency issue. Assume you keep the list in alphabetical order. Then when we find the first matching element there is no need to traverse the remainder of the list, although the `member` predicate above will always do so.

So what do we do if we want to only check membership, or find the first occurrence of an element in a list? Unfortunately, there is no easy answer, because the most straighforward solution

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{X \neq Y \quad \mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

requires disequality which is problematic in the presence of variables. In Prolog notation:

```
member1(X, [X|Ys]).
member1(X, [Y|Ys]) :- X \= Y, member1(X, Ys).
```

When both arguments are ground, this works as expected, giving just one solution to the query

```
?- member1(a, [a,b,a,c]).
```

However, when we ask

```
?- member1(X, [a,b,a,c]).
```

we only get one answer, namely `X = a`. The reason is that when we come to the second clause, we instantiate `Y` to `a` and `Ys` to `[b,a,c]`, and the body of the clause becomes

```
X \= a, member1(X, [b,a,c]).
```

Now we have the problem that we cannot determine if `X` is different from `a`, because `X` is still a variable. Prolog interprets $s \neq t$ as *non-unifiability*, that is, $s \neq t$ succeeds of s and t are not unifiable. But `X` and `a` are unifiable, so the subgoal fails and no further solutions are generated.[1]

There are two attitudes we can take. One is to restrict the use of disequality (and, therefore, here also the use of `member1`) to the case where both sides have no variables in them. In that case disequality can be easily checked without problems. This is the solution adopted by Prolog, and one which we adopt for now.

The second one is to postpone the disequality $s \neq t$ until we can tell from the structure of s and t that they will be different (in which case we succeed) or the same (in which case the disequality fails). The latter solution requires a much more complicated operational semantics because some goals must be postponed until their arguments become instantiated. This is the general topic of *constructive negation*[2] [1] in the setting of *constraint logic programming* [2, 6], which we will address later in the course.

Disequality is related to the more general question of negation, because $s \neq t$ is the negation of equality, which is a simple predicate that is either primitive, or could be defined with the one clause `X = X`.

[1] One must remember, however, that in Prolog unification is not sound because it omits the occurs-check, as hinted at in the previous lecture. This also affects the correctness of disequality.

[2] The use of the word "constructive" here is unrelated to its use in logic.

2.5 Simple List Predicates

After the member predicate generated some interesting questions, we explore some other list operations. We start with prefix(xs, ys) which is supposed to hold when the list xs is a prefix of the list ys. The definition is relatively straightforward.

```
prefix([], Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).
```

Conversely, we can test for a suffix.

```
suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :- suffix(Xs, Ys).
```

Interestingly, these predicates can be used in a variety of ways. We can check is one list is a prefix of another, we can enumerate prefixes, and we can even enumerate prefixes and lists. For example:

```
?- prefix(Xs,[a,b,c,d]).
Xs = [];
Xs = [a];
Xs = [a,b];
Xs = [a,b,c];
Xs = [a,b,c,d].
```

enumerates all prefixes, while

```
?- prefix(Xs,Ys).
Xs = [];

Xs = [A]
Ys = [A|_];

Xs = [A,B]
Ys = [A,B|_];

Xs = [A,B,C]
Ys = [A,B,C|_];

Xs = [A,B,C,D]
Ys = [A,B,C,D|_];
...
```

enumerates lists together with prefixes. Note that `A`, `B`, `C`, and `D` are variables, as is the underscore `_` so that for example `[A|_]` represents any list with at least one element.

A more general prediate is append(xs, ys, zs) which holds when zs is the result of appending xs and ys.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

append can also be used in different directions, and we can also employ it for alternative definitions of prefix and suffix.

```
prefix2(Xs, Ys) :- append(Xs, _, Ys).
suffix2(Xs, Ys) :- append(_, Xs, Ys).
```

Here we have used anonymous variables '_'. Note that when several underscores appear in a clauses, each one stands for a different anonymous variable. For example, if we want to define a sublist as a suffix of a prefix, we have to name the intermediate variable instead of leaving it anonymous.

```
sublist(Xs, Ys) :- prefix(Ps, Ys), suffix(Xs, Ps).
```

This definition of `sublist` has some shortcomings (see Exercise 2.1).

2.6 Sorting

As a slightly larger example, we use a recursive definition of quicksort. This is particularly instructive as it clarifies the difference between a specification and an implemention. A specification for sort(xs, ys) would simply say that ys is an ordered permutation of xs. However, this specification is not useful as an implementation: we do not want to cycle through all possible permutations until we find one that is ordered.

Instead we implement a non-destructive version of quicksort, modeled after similar implementations in functional programming. We use here the built-in Prolog integers, rather than the unary representation from the previous lecture. Prolog integers can be compared with n `=<` m (n is less or equal to m) and n `>` m (n is greater than m) and similar predicates, written in infix notation. In order for these comparisons to make sense, the arguments must be instantiated to actual integers and are not allowed to be variables, which constitute a run-time error. This combines two conditions: one which is called a *mode* is that `=<` and `<` require their arguments

to be *ground* upon invocation, that is not contain any variables. The second condition is a type condition which requires the arguments to be integers. Since these conditions can not be enforced at compile time, they are signaled as run-time errors.

Quicksort proceeds by partitioning the tail of the input list into those elements that are smaller than or equal to its first element and those that are larger than its first element. It then recursively sorts the two sublists and appends the results.

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

Partitioning a list about the pivot element `X0` is also straightforward.

```
partition([], _, [], []).
partition([X|Xs], X0, [X|Ls], Gs) :-
        X =< X0, partition(Xs, X0, Ls, Gs).
partition([X|Xs], X0, Ls, [X|Gs]) :-
        X > X0, partition(Xs, X0, Ls, Gs).
```

Note that the second and third case are both guarded by comparisons. This will fail if either `X` or `X0` are uninstantiated or not integers. The predicate `partition(`xs`,` x_0`,` ls`,` gs`)` therefore inherits a mode and type restriction: the first argument must be a ground list of integers and the second argument must be a ground integer. If these conditions are satisfied and `partition` succeeds, the last two arguments will always be lists of ground integers. In a future lecture we will discuss how to enforce conditions of this kind to discover bugs early. Here, the program is small, so we can get by without mode checking and type checking.

It may seem that the check `X > X0` in the last clause is redundant. However, that is not the case because upon backtracking we might select the second clause, even if the first one succeeded earlier, leading to an incorrect result. For example, without this guard the query

```
?- quicksort([2,1,3], Ys)
```

would incorrectly return `Ys = [2,1,3]` as its second solution.

In this particular case, the test is trivial so the overhead is acceptable. Sometimes, however, a clause is guarded by a complicated test which takes a long time to evaluate. In that case, there is no easy way to avoid evaluating it twice, in pure logic programming. Prolog offers several ways to work around this limitation which we discuss in the next section.

2.7 Conditionals

We use the example of computing the minimum of two numbers as an example analogous to `partition`, but shorter.

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- X > Y.
```

In order to avoid the second, redundant test we can use Prolog's *conditional* construct, written as

```
A -> B ; C
```

which solves goal A. If A succeeds we commit to the solution, removing all choice points created during the search for a proof of A and then solve B. If A fails we solve C instead. There is also a short form `A -> B` which is equivalent to `A -> B ; fail` where `fail` is a goal that always fails.

Using the conditional, `minimum` can be rewritten more succinctly as

```
minimum(X, Y, Z) :- X =< Y -> Z = X ; Z = Y.
```

The price that we pay here is that we have to leave the realm of pure logic programming.

Because the conditional is so familiar from imperative and functional program, there may be a tendency to overuse the conditional when it can easily be avoided.

2.8 Cut

The conditional combines two ideas: commiting to all choices so that only the first solution to a goal will be considered, and branching based on that first solution.

A more powerful primitive is *cut*, written as '`!`', which is unrelated to the use of the word "cut" in proof theory. A cut appears in a goal position and commits to all choices that have been made since the clause it appears in has been selected, including the choice of that clause. For example, the following is a correct implementation of `minimum` in Prolog.

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y).
```

The first clause states that if x is less or equal to y than the minimum is equal to x. Moreover, we commit to this clause in the definition of minimum and on backtracking we do not attempt to use the second clause (which would otherwise be incorrect, of course).

If we permit meta-calls in clauses, then we can define the conditional `A -> B ; C` using cut with

```
if_then_else(A, B, C) :- A, !, B.
if_then_else(A, B, C) :- C.
```

The use of cut in the first clause removes all choice points created during the search for a proof of A when it succeeds for the first time, and also commits to the first clause of `if_then_else`. The solution of B will create choice points and backtrack as usual, except when it fails the second clause of `if_then_else` will never be tried.

If A fails before the cut, then the second clause will be tried (we haven't committed to the first one) and C will be executed.

Cuts can be very tricky and are the source of many errors, because their interpretation depends so much on the operational behavior of the program rather than the logical reading of the program. One should resist the temptation to use cuts excessively to improve the efficiency of the program unless it is truly necessary.

Cuts are generally divided into *green cuts* and *red cuts*. Green cuts are merely for efficiency, to remove redundant choice points, while red cuts change the meaning of the program entirely. Revisiting the earlier code for `minimum` we see that it is a red cut, since the second clause does not make any sense by itself, but only because the the first clause was attempted before. The cut in

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y) :- X > Y.
```

is a green cut: removing the cut does not change the meaning of the program. It still serves some purpose here, however, because it prevents the second comparison to be carried out if the first one succeeds (although it is still performed redundantly if the first one fails).

A common error is exemplified by the following attempt to make the `minimum` predicate more efficient.

```
% THIS IS INCORRECT CODE
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

At first this seems completely plausible, but it is nonetheless incorrect. Think about it before you look at the counterexample at the end of these notes—it is quite instructive.

2.9 Negation as Failure

One particularly interesting use of cut is to implement negation as finite failure. That is, we say that A is false if the goal A fails. Using higher-order techniques and we can implement `\+(A)` with

```
\+(A) :- A, !, fail.
\+(A).
```

The second clause seems completely contrary to the definition of negation, so we have to interpret this program operationally. To solve `\+(`A`)` we first try to solve A. If that fails we go the second clause which always succeeds. This means that if A fails then `\+(`A`)` will succeed without instantiating any variables. If A succeeds then we commit and fail, so the second clause will never be tried. In this case, too, no variables are instantiated, this time because the goal fails.

One of the significant problem with negation as failure is the treatment of variables in the goal. That is, `\+(`A`)` succeeds if there is no instance of A that is true. On the other hand, it fails if there is an instance of A that succeeds. This means that free variables may not behave as expected. For example, the goal

```
?- \+(X = a).
```

will fail. According the usual interpretation of free variables this would mean that there is no term t such that $t \neq a$ for the constant a. Clearly, this interpretation is incorrect, as, for example,

```
?- \+(b = a).
```

will succeed.

This problem is similar to the issue we identified for disequality. When goals may not be ground, negation as failure should be viewed with distrust and is probably wrong more often than it is right.

There is also the question on how to reason about logic programs containing disequality, negation as failure, or cut. I do not consider this to be a solved research question.

2.10 Prolog Arithmetic

As mentioned and exploited above, integers are a built-in data type in Prolog with some predefined predicates such as =< or >. You should consult your Prolog manual for other built-in predicates. There are also some built-in operations such as addition, subtraction, multiplication, and division. Generally these operations can be executed using a special goal of the form t `is` e which evaluates the arithmetic expression e and unifies the result with term t. If e cannot be evaluated to a number, a run-time error will result. As an example, here is the definition of the length predicate for Prolog using built-in integers.

```
% length(Xs, N) iff Xs is a list of length N.
length([], 0).
length([X|Xs], N) :- length(Xs, N1), N is N1+1.
```

As is often the case, the left-hand side of the `is` predicate is a variable, the right-hand side an expression. Of course, these variables cannot be updated destructively.

2.11 Exercises

Exercise 2.1 *Identify a shortcoming of the given definition of* sublist *and rewrite it to avoid the problem.*

Exercise 2.2 *Write specifications* ordered(xs) *to check if the list xs of integers (as built into Prolog) is in ascending order, and* permutation(xs, ys) *to check if ys is a permutation of xs. Write a possibly very slow implementation of* sort(xs, ys) *to check if ys is an ordered permutation of xs. It should be usable to sort a given list xs.*

Exercise 2.3 *Quicksort is not the fastest way to sort in Prolog, which is reputedly mergesort. Write a logic program for* mergesort(xs, ys) *for lists of integers as built into Prolog.*

Exercise 2.4 *The Dutch national flag problem is to take a list of elements that are either red, white, or blue and return a list with all red elements first, followed by all white elements, with all blue elements last (the order in which they appear on the Dutch national flag). We represent the property of being red, white, or blue with three predicates,* `red`(x), `white`(x), *and* `blue`(x). *You may assume that every element of the input list satisfies exactly one of these three predicates.*

Write a Prolog program to solve the Dutch national flag problem. Try to take advantage of the intrinsic expressive power of logic programming to obtain an elegant program.

2.12 Answer

The problem is that a query such as

```
?- minimum(5,10,10).
```

will succeed because is fails to match the first clause head.

The general rule of thumb is to leave output variables (here: in the third position) unconstrained free variables and unify it with the desired output after the cut. This leads to the earlier version of `minimum` using cut.

2.13 References

[1] D. Chan. Constructive negation based on the complete database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICSLP'88)*, pages 111–125, Seattle, Washington, September 1988. MIT Press.

[2] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[3] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[4] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[5] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[6] Peter J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

15-819K: Logic Programming

Lecture 3

Induction

Frank Pfenning

September 5, 2006

One of the reasons we are interested in high-level programming languages is that, if properly designed, they allow rigorous reasoning about properties of programs. We can prove that our programs won't crash, or that they terminate, or that they satisfy given specifications. Logic programs are particularly amenable to formal reasoning.

In this lecture we explore induction, with an emphasis on *induction on the structure of deductions* sometimes called *rule induction* in order to prove properties of logic programs.

3.1 From Logical to Operational Meaning

A logic program has multiple interpretations. One is as a set of inference rules to deduce logical truths. Under this interpretation, the order in which the rules are written down, or the order in which the premisses to a rule are listed, are completely irrelevant: the true propositions and even the structure of the proofs remain the same. Another interpretation is as a program, where proof search follows a fixed strategy. As we have seen in prior lectures, both the order of the rules and the order of the premisses of the rules play a significant role and can make the difference between a terminating and a non-terminating computation and in the order in which answer substitutions are returned.

The different interpretations of logic programs are linked. The strength of that link depends on the presence or absence of purely operational constructs such as conditionals or cut, and on the details of the operational semantics that we have not yet discussed.

The most immediate property is *soundness* of the operational semantics: if a query A succeeds with a substitution θ, then the result of applying the

substitution θ to A (written $A\theta$) is true under the logical semantics. In other words, $A\theta$ has a proof. This holds for pure logic programs but does not hold in the presence of logic variables together with negation-as-failure, as we have seen in the last lecture.

Another property is *completeness* of the operational semantics: if there is an instance of the query A that has a proof, then the query should succeed. This does not hold, since logic programs do not necessarily terminate even if there is a proof.

But there are some intermediate points. For example, the property of *non-deterministic completeness* says that if the interpreter were always allowed to choose which rule to use next rather than having to use the first applicable one, then the interpreter would be complete. Pure logic programs are complete in this sense. This is important because it allows us to interpret finite failure as falsehood: if the interpreter returns with the answer '`no`' it has explored all possible choices. Since none of them has led to a proof, and the interpreter is non-deterministically complete, we know that no proof can exist.

Later in the course we will more formally establish soundness and non-deterministic completeness for pure logic programs. It is relevant for this lecture, because when we want to reason about logic programs it is important to consider at which level of abstraction this reasoning takes place: Do we consider the logical meaning? Or the operational meaning including the backtracking behavior? Or perhaps the non-deterministic operational meaning? Making a mistake here could lead to a misinterpretation of the theorem we proved, or to a large amount of unnecessary work. We will point out such consequences as we go through various forms of reasoning.

3.2 Rule Induction

We begin by reasoning about the logical meaning of programs. As a simple example, we go back to the unary encoding of natural numbers from the first lecture. For reference we repeat the predicates for even and plus

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evs}$$

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

Our aim is to prove that the sum of two even numbers is even. It is not

immediately obvious how we can express this property on the relational specification. For example, we might say:

> *For any* m, n, *and* p, *if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *and* $\mathsf{plus}(m, n, p)$ *then* $\mathsf{even}(p)$.

Or we could expressly require the existence of a sum p and the fact that it is even:

> *For any* m, n, *if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *then there exists a* p *such that* $\mathsf{plus}(m, n, p)$ *and* $\mathsf{even}(p)$.

If we knew that plus is a total function in its first two arguments (that is, "*For any* m *and* n *there exists a unique* p *such that* $\mathsf{plus}(m, n, p)$."), then these two would be equivalent (see Exercise 3.2).

We will prove it in the second form. The first idea for this proof is usually to examine the definition of plus and see that it is defined structurally over its first argument m: the rule pz accounts for z and the rule ps reduces $\mathsf{s}(m)$ to m. This suggests an induction over m. However, in the predicate calculus (and therefore also in our logic programming language), m can be an arbitrary term and is therefore not a good candidate for induction.

Looking at the statement of the theorem, we see we are given the information that $\mathsf{even}(m)$. This means that we have a deduction of $\mathsf{even}(m)$ using only the two rules evz and evs, since we viewed these two rules as a complete definition of the predicate $\mathsf{even}(m)$. This licenses us to proceed by induction on the structure of the deduction of $\mathsf{even}(m)$. This is sometimes called *rule induction*. If we want to prove a property for all proofs of a judgment, we consider each rule in turn. We may assume the property for all premisses of the rule and have to show that it holds for the conclusion. If we can show this for all rules, we know the property must hold for all deductions.

In our proofs, we will need names for deductions. We use script letters $\mathcal{D}$, $\mathcal{E}$, and so on, to denote deduction and use the two-dimensional notation

$$\begin{array}{c} \mathcal{D} \\ J \end{array}$$

if $\mathcal{D}$ is a deduction of J.

Theorem 3.1 *For any* m, n, *if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *then there exists a* p *such that* $\mathsf{plus}(m, n, p)$ *and* $\mathsf{even}(p)$.

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $\mathsf{even}(m)$.

Case: $\mathcal{D} = \dfrac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz}$ where $m = \mathsf{z}$.

$\mathsf{even}(n)$	Assumption
$\mathsf{plus}(\mathsf{z}, n, n)$	By rule pz
There exists p such that $\mathsf{plus}(\mathsf{z}, n, p)$ and $\mathsf{even}(p)$	Choosing $p = n$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(m')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}\ \mathsf{evs}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

$\mathsf{even}(n)$	Assumption
$\mathsf{plus}(m', n, p')$ and $\mathsf{even}(p')$ for some p'	By ind. hyp. on $\mathcal{D}'$
$\mathsf{plus}(\mathsf{s}(m'), n, \mathsf{s}(p'))$	By rule ps
$\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, \mathsf{s}(\mathsf{s}(p')))$	By rule ps
$\mathsf{even}(\mathsf{s}(\mathsf{s}(p')))$	By rule evs
There exists p such that $\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$ and $\mathsf{even}(p)$	Choosing $p = \mathsf{s}(\mathsf{s}(p'))$.

□

We have written here the proof in each case line-by-line, with a justification on the right-hand side. We will generally follow this style in this course, and you should arrange the answers to exercises in the same way because it makes proofs relatively easy to check.

3.3 Deductions and Proofs

One question that might come to mind is: Why did we have to carry out an inductive proof by hand in the first place? Isn't logic programming proof search according to a fixed strategy, so can't we get the operational semantics to do this proof for us?

Unfortunately, logic programming search has some severe restrictions so that it is usable as a programming language and has properties such as soundness and non-deterministic completeness. The restrictions are placed both on the forms of programs and the forms of queries. So far, in the logic that underlies Prolog, rules establish only atomic predicates. Furthermore, we can only form queries that are conjunctions of atomic propositions, possibly with some variables. This means that queries are purely *existential*: we asked whether there *exists* some instantiation of the variables

so that there *exists* a proof for the resulting proposition as in the query `?- plus(s(z), s(s(z)), P)` where we simultaneously ask for a p and a proof of $\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), p)$.

On the other hand, our theorem above is primarily *universal* and only on the inside do we see an *existential* quantifier: "*For every* m *and* n*, and for every deduction* $\mathcal{D}$ *of* $\mathsf{even}(m)$ *and* $\mathcal{E}$ *of* $\mathsf{even}(n)$ *there exists a* p *and deductions* $\mathcal{F}$ *of* $\mathsf{plus}(m, n, p)$ *and* $\mathcal{G}$ *of* $\mathsf{even}(p)$*.*"

This difference is also reflected in the structure of the proof. In response to a logic programming query we only use the inference rules defining the predicates directly. In the proof of the theorem about addition, we instead use induction in order to show that deductions of $\mathsf{plus}(m, n, p)$ and $\mathsf{even}(p)$ exist. If you carefully look at our proof, you will see that it contains a recipe for constructing these deductions from the given ones, but it does not construct them by backward search as in the operational semantics for logic programming. As we will see later in the course, it is in fact possible to represent the induction proof of our first theorem also in logic programming, although it cannot be found only by logic programming search.

We will make a strict separation between proofs using only the inference rules presented by the logic programmer and proofs *about* these rules. We will try to be consistent and write *deduction* for a proof constructed directly with the rules and *proof* for an argument about the logical or operational meaning of the rules. Similarly, we reserve the terms *proposition*, *goal*, and *query* for logic programs, and *theorem* for properties of logic programs.

3.4 Inversion

An important step in many induction proofs is *inversion*. The simplest form of inversion arises if have established that a certain proposition is true, and that the proposition matches the conclusion of only one rule. In that case we know that this rule must have been used, and that all premisses of the rule must also be true. More generally, if the proposition matches the conclusion of several rules, we can split the proof into cases, considering each one in turn.

However, great care must be taken with applying inversion. In my experience, the most frequent errors in proofs, both by students in courses such as this and in papers submitted to or even published in journals, are (a) missed cases that should have been considered, and (b) incorrect applications of inversion. We can apply inversion only if we already know that a judgment has a deduction, and then we have to take extreme care to make sure that we are indeed considering all cases.

As an example we prove that the list difference is uniquely determined, if it exists. As a reminder, the definition of append in rule form. We use the Prolog notation $[\,]$ for the empty list, and $[x|xs]$ for the list with head x and tails xs.

$$\frac{}{\mathsf{append}([\,], ys, ys)}\ \mathsf{apnil} \qquad \frac{\mathsf{append}(xs, ys, zs)}{\mathsf{append}([x|xs], ys, [x|zs])}\ \mathsf{apcons}$$

We express this in the following theorem.

Theorem 3.2 *For all xs and zs and for all ys and ys', if* $\mathsf{append}(xs, ys, zs)$ *and* $\mathsf{append}(xs, ys', zs)$ *then $ys = ys'$.*

Proof: By induction on the deduction $\mathcal{D}$ of $\mathsf{append}(xs, ys, zs)$. We use $\mathcal{E}$ to denote the given deduction $\mathsf{append}(xs, ys', zs)$.

Case: $\mathcal{D} = \dfrac{}{\mathsf{append}([\,], ys, ys)}$ where $xs = [\,]$ and $zs = ys$.

$\mathsf{append}([\,], ys', ys)$	Given deduction $\mathcal{E}$
$ys' = ys$	By inversion on $\mathcal{E}$ (rule apnil)

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{append}(xs_1, ys, zs_1)\end{array}}{\mathsf{append}([x|xs_1], ys, [x|zs_1])}$ where $xs = [xs|xs_1]$, $zs = [xs|zs_1]$.

$\mathsf{append}([x|xs_1], ys', [x|zs_1])$	Given deduction $\mathcal{E}$
$\mathsf{append}(xs_1, ys', zs_1)$	By inversion on $\mathcal{E}$ (rule apcons)
$ys = ys'$	By ind. hyp. on $\mathcal{D}_1$

□

3.5 Operational Properties

We do not yet have formally described the operational semantics of logic programs. Therefore, we cannot prove operational properties completely rigorously, but we can come close by appealing to the intuitive semantics. Consider the following perhaps somewhat unfortunate specification of the predicate digit for decimal digits in unary notation, that is, natural numbers between 0 and 9.

$$\frac{}{\mathsf{digit}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))))))} \qquad \frac{\mathsf{digit}(\mathsf{s}(N))}{\mathsf{digit}(N)}$$

For example, we can deduce that z is a digit by using the second rule nine times (working bottom up) and then closing of the deduction with the first rule. In Prolog notation:

```
digit(s(s(s(s(s(s(s(s(s(z))))))))).
digit(N) :- digit(s(N)).
```

While logically correct, this does not work correctly as a decision procedure, because it will not terminate for any argument greater than 9.

Theorem 3.3 *Any query* `?- digit(`n`)` *for* $n > 9$ *will not terminate.*

Proof: By induction on the computation. If $n > 9$, then the first clause cannot apply. Therefore, the goal `digit(`n`)` must be reduced to the subgoal `digit(s(`n`))` by the second rule. But $\mathsf{s}(n) > 9$ if $n > 9$, so by induction hypothesis the subgoal will not terminate. Therefore the original goal also does not terminate. □

3.6 Aside: Forward Reasoning and Saturation

As mentioned in the first lecture, there is a completely different way to interpret inference rules as logic programs than the reading that underlies Prolog. This idea is to start with axioms (that is, inference rules with no premisses) as logical truths and apply all rules in the forward direction, adding more true propositions. We stop when any rule application that we could perform does not change the set of true propositions. In that case we say the database of true propositions is *saturated*. In order to answer a query we can now just look it up in the saturated database: if an instance of the query is in the database, we succeed, otherwise we fail.

In the example from above, we start with a database consisting only of $\mathsf{digit}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))))))$. We can apply the second rule with this as a premiss to conclude that $\mathsf{digit}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))))))$. We can repeat this process a few more times until we finally conclude $\mathsf{digit}(\mathsf{z})$. At this point, any further rule applications would only add facts with already know: the set is saturated. We see that, consistent with the logical meaning, only the numbers 0 through 9 are digits, other numbers are not.

In this example, the saturation-based operational semantics via forward reasoning worked well for the given rules, while backward reasoning did not. There are classes of algorithms which appear to be easy to describe via saturation, that appear significantly more difficult with backward reasoning and vice versa. We will therefore return to forward reasoning and

saturation later in the class, and also consider how it may be integrated with backward reasoning in a logical way.

3.7 Historical Notes

The idea to mix reasoning *about* rules with the usual logic programming search goes back to work by Eriksson and Hallnäs [2] which led to the GCLA logic programming language [1]. However, it stops short of supporting full induction. More recently, this line of development been revived by Tiu, Nadathur, and Miller [9]. Some of these ideas are embodied in the Bedwyr system currently under development.

Another approach is to keep the layers separate, but provide means to express proofs of properties of logic programs again as logic programs, as proposed by Schürmann [8]. These ideas are embodied in the Twelf system [6].

Saturating forward search has been the mainstay of the theorem proving community since the pioneering work on resolution by Robinson [7]. In logic programming, it has been called *bottom-up evaluation* and has historically been applied mostly in the context of databases [5] where saturation can often be guaranteed by language restrictions. Recently, it has been revisited as a tool for algorithm specification and complexity analysis by Ganzinger and McAllester [3, 4].

3.8 Exercises

The proofs requested below should be given in the style presented in these notes, with careful justification for each step of reasoning. If you need a lemma that has not yet been proven, carefully state and prove the lemma.

Exercise 3.1 *Prove that the sum of two even numbers is even in the first form given in these notes.*

Exercise 3.2 *Prove that* $\mathsf{plus}(m, n, p)$ *is a total function of its first two arguments and exploit this property to prove carefully that the two formulations of the property that the sum of two even numbers is even, are equivalent.*

Exercise 3.3 *Prove that* $\mathsf{times}(m, n, p)$ *is a total function of its first two arguments.*

Exercise 3.4 *Give a relational interpretation of the claim that "addition is commutative" and prove it.*

Exercise 3.5 *Prove that for any list* xs, append$(xs, [\,], xs)$.

Exercise 3.6 *Give two alternative relational interpretations of the statement that "*append *is associative." Prove one of them.*

Exercise 3.7 *Write a logic program to reverse a given list and prove that when reversing the reversed list, we obtain the original list.*

Exercise 3.8 *Prove the correctness of quicksort from the previous lecture with respect to the specification from Exercise 2.2: If* `quicksort(`xs`,` ys`)` *is true then the second argument is an ordered permutation of the first. Your proof should be with respect to logic programs to check whether a list is ordered, and whether one list is a permutation of another.*

3.9 References

[1] M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin. The programming language GCLA—a definitional approach to logic programming. *New Generation Computing*, 7(4):381–404, 1990.

[2] Lars-Henrik Eriksson and Lars Hallnäs. A programming calculus based on partial inductive definitions. SICS Research Report R88013, Swedish Institute of Computer Science, 1988.

[3] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T.Nipkow R.Goré, A.Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

[4] Harald Ganzinger and David A. McAllester. Logical algorithms. In P. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[5] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson*, pages 640–700. MIT Press, Cambridge, Massachusetts, 1991.

[6] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[7] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[8] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[9] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In C.Benzmüller, J.Harrison, and C.Schürmann, editors, *Proceedings of the Workshop on Empirically Successful Automated Reasnoing in Higher-Order Logics (ESHOL'05)*, pages 79–98, Montego Bay, Jamaica, December 2005.

15-819K: Logic Programming

Lecture 4

Operational Semantics

Frank Pfenning

September 7, 2006

In this lecture we begin in the quest to formally capture the operational semantics in order to prove properties of logic programs that depend on the way proof search proceeds. This will also allow us to relate the logical and operational meaning of programs to understand deeper properties of logic programs. It will also form the basis to prove the correctness of various form of program analysis, such as type checking or termination checking, to be introduced later in the class.

4.1 Explicating Choices

To span the distance between the logical and the operational semantics we have to explicate a series of choices that are fixed when proof search proceeds. We will proceed in this order:

1. Left-to-right subgoal selection. In terms of inference rules, this means that we first search for a proof of the first premiss, then the second, etc.

2. First-to-last clause selection and backtracking. In terms of inference rules this means when more than one rule is applicable, we begin by trying the one listed first, then the one listed second, etc.

3. Unification. In terms of inference rules this means when we decide how to instantiate the schematic variables in a rule and the unknowns in a goal, we use a particular algorithm to find the most general unifier between the conclusion of the rule and the goal.

4. Cut. This has no reflection at the level of inference rules. We have to specify how we commit to particular choices made so far when we encounter a cut or another control constructs such as a conditional.

5. Other built-in predicates. Prolog has other built-in predicates for arithmetic, input and output, changing the program at run-time, foreign function calls, and more which we will not treat formally.

It is useful not to jump directly to the most accurate and low-level semantics, because we often would like to reason about properties of programs that are independent of such detail. One set of examples we have already seen: we can reason about the logical semantics to establish properties such as that the sum of two even numbers is even. In that case we are only interested in successful computations, and how we searched for them is not important. Another example is represented by cut: if a program does not contain any cuts, the complexity of the semantics that captures it is unwarranted.

4.2 Definitional Intpreters

The general methodology we follow goes back to Reynolds [3], adapted here to logic programming. We can write an interpreter for a language in the language itself (or a very similar language), a so-called *definitional interpreter*, *meta-interpreter*, or *meta-circular interpreter*. This may fail to completely determine the behavior of the language we are studying (the *object language*), because it may depend on the behavior of the language in which we write the definition (the *meta-language*), and the two are the same! We then transform the definitional interpreter, removing some of the advanced features of the language we are defining, so that now the more advanced constructs are explained in terms of simpler ones, removing circular aspects. We can interate the process until we arrive at the desired level of specification.

For Prolog (although not pure first-order logic programming), the simplest meta-interpreter, hardly deserving the name, is

```
solve(A) :- A.
```

To interpret the argument to `solve` as a goal, we simply execute it using the meta-call facility of Prolog.

This does not provide a lot of insight, but it brings up the first issue: how do we represent logic programs and goals in order to execute them

in our definitional interpreter? In Prolog, the answer is easy: we think of the comma which separates the subgoal of a clause as a binary function symbol denoting conjunction, and we think of the constant `true` which always succeeds as just a constant. One can think of this as replicating the language of predicates in the language of function symbols, or not distinguishing between the two. The code above, if it were formally justified using higher-order logic, would take the latter approach: logical connectives *are* data and can be treated as such. In the next interpreter we take the first approach: we overload comma to separate subgoals in the meta-language, but we also use it as a function symbol to describe conjunction in the object language. Unlike in the code above, we will not mix the two. The logical constant `true` is similarly overloaded as a predicate constant of the same name.

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

In the second clause, the head `solve((A , B))` employs infix notation, and could be written equivalently as `solve(','(A, B))`.[1] The additional pair of parentheses is necessary since `solve(A , B)` would be incorrectly seen as a predicate `solve` with two arguments.

The predicate `clause/2` is a built-in predicate of Prolog.[2] The subgoal `clause(P, B)` will unify `P` with the head of each program clause and `B` with the corresponding body. In other words, if `clause(P, B)` succeeds, then `P :- B.` is an instance of a clause in the current program. Prolog will try to unify `P` and `B` with the clauses of the current program first-to-last, so that the above meta-interpreter will work correctly with respect to the intuitive semantics explained earlier.

There is a small amount of standardization in that a clause P`.` in the program with an empty body is treated as if it were P `:- true.`

This first interpreter does not really explicate anything: the order in which subgoals are solved in the object language is the same as the order in the meta-language, according to the second clause. The order in which clauses are tried is the order in which `clause/2` delivers them. And unification between the goal and the clause head is also inherited by the object

[1] In Prolog, we can quote an infix operator to use it as an ordinary function or predicate symbol.

[2] In Prolog, it is customary to write p/n when refering to a predicate p of arity n, since many predicates have different meanings at different arities.

language from the meta-language through the unification carried out by `clause(P, B)` between its first argument and the clause heads in the program.

4.3 Subgoal Order

According to our outline, the first task is to modify our interpreter so that the order in which subgoals are solved is made explicit. When encountering a goal `(`A `,` B`)` we push B onto a stack and solve A first. When A as has been solved we then pop B off the stack and solve it. We could represent the stack as a list, but we find it somewhat more elegant to represent the goal stack itself as a conjunction of goals because all the elements of goal stack have to be solved for the overall goal to succeed.

The `solve` predicate now takes two arguments, `solve(`A`,` S`)` where A is the goal, and S is a stack of yet unsolved goals. We start with the empty stack, represented by `true`.

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

We explain each clause in turn.
If the goal is solved and the goal stack is empty, we just succeed.

```
solve(true, true).
```

If the goal is solved and the goal stack is non-empty, we pop the most recent subgoal A of the stack and solve it.

```
solve(true, (A , S)) :- solve(A, S).
```

If the goal is a conjunction, we solve the first conjunct, pushing the second one on the goal stack.

```
solve((A , B), S) :- solve(A, (B , S)).
```

When the goal is atomic, we match it against the heads of all clauses in turn, solving the body of the clause as a subgoal.

```
solve(P, S) :- clause(P, B), solve(B, S).
```

We do not explicitly check that P is atomic, because `clause(`P`,` B`)` will fail if it is not.

4.4 Subgoal Order More Abstractly

The interpreter from above works for pure Prolog as intended. Now we would like to prove properties of it, such as its soundness: if it proves `solve(`A`, true)` then it is indeed the case that A is true. In order to do that it is advisable to reconstruct the interpreter above in *logical* form, so we can use induction on the structure of deductions in a rigorous manner.

The first step is to define our first logical connective: conjunction! We also need a propositional constant denoting truth. It is remarkable that for all the development of logic programming so far, not a single logical connective was needed, just atomic propositions, the truth judgment, and deductions as evidence for truth.

When defining logical connectives we follow exactly the same ideas as for defining atomic propositions: we define them via inference rules, specifying what counts as evidence for their truth.

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge I \qquad\qquad \frac{}{\top\ \mathit{true}}\ \top I$$

These rules are called *introduction rules* because they introduce a logical connective in the conclusion.

Next we define a new judgment on propositions. Unlike $A\ \mathit{true}$ this is a binary judgment on two propositions. We write it $A\ /\ S$ and read it as A *under* S. We would like it to capture the logical form of `solve(`A`, `S`)`. The first three rules are straightforward, and revisit the corresponding rules for `solve/2`.

$$\frac{}{\top\ /\ \top} \qquad\qquad \frac{A\ /\ S}{\top\ /\ A \wedge S}$$

$$\frac{A\ /\ B \wedge S}{A \wedge B\ /\ S}$$

The last "rule" is actually a whole family of rules, one for each rule about truth of atoms P.

$$\frac{B_1 \wedge \cdots \wedge B_m\ /\ S}{P\ /\ S} \qquad \text{for each rule} \qquad \frac{B_1\ \mathit{true} \ldots B_m\ \mathit{true}}{P\ \mathit{true}}$$

We write the premiss as $\top\ /\ S$ if $m = 0$, thinking of $\top$ as the empty conjunction.

It is important that the definitions of truth ($A\ jtrue$) and provability under a stack-based search strategy ($A\ /\ S$) do not mutually depend on each other so we can relate them cleanly.

Note that each rule of the new judgment $A\ /\ S$ has either one or zero premisses. In other words, if we do proof search via goal-directed search, the question of subgoal order does not arise. It has explicitly resolved by the introduction of a subgoal stack. We can now think of these rules as just defining a transition relation, reading each rule from the conclusion to the premiss. This transition relation is still non-deterministic, because more than one rule could match an atomic predicate, but we will resolve this as well as we make other aspects of the semantics more explicit.

4.5 Soundness

To show the soundness of the new judgment with respect to truth, we would like to show that if $A\ /\ \top$ then $A\ true$. This, however, is not general enough to prove by induction, since if A is a conjunction, the premiss will have a non-empty stack and the induction hypothesis will not be applicable. Instead we generalize the induction hypothesis. This is usually a very difficult step; in this case, however, it is not very difficult to see what the generalization should be.

Theorem 4.1 *If $A\ /\ S$ then $A\ true$ and $S\ true$.*

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $A\ /\ S$.

Case: $\mathcal{D} = \dfrac{}{\top\ /\ \top}$ where $A = S = \top$.

$A\ true$	By $A = \top$ and rule $\top I$
$S\ true$	By $S = \top$ and rule $\top I$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2 \\ A_1\ /\ S_2\end{array}}{\top\ /\ A_1 \wedge S_2}$ where $A = \top$ and $S = A_1 \wedge S_2$.

$A\ true$	By $A = \top$ and rule $\top I$
$A_1\ true$ and $S_2\ true$	By ind.hyp. on $\mathcal{D}_2$
$A_1 \wedge S_2\ true$	By rule $\wedge I$
$S\ true$	Since $S = A_1 \wedge S_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ A_1 \mid A_2 \wedge S\end{array}}{A_1 \wedge A_2 \mid S}$ where $A = A_1 \wedge A_2$.

$A_1\ true$ and $A_2 \wedge S\ true$	By ind.hyp. on $\mathcal{D}_1$
$A_2\ true$ and $S\ true$	By inversion (rule $\wedge I$)
$A_1 \wedge A_2\ true$	By rule $\wedge I$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ B_1 \wedge \cdots \wedge B_m \mid S\end{array}}{P \mid S}$ where $A = P$ and $\dfrac{B_1\ true \ldots B_m\ true}{P\ true}$.

$B_1 \wedge \cdots \wedge B_m\ true$ and $S\ true$	By ind.hyp. on $\mathcal{D}'$
$B_1\ true, \ldots, B_m\ true$	By $m-1$ inversion steps if $m > 0$ ($\wedge I$)
$P\ true$	By given inference rule

If $m = 0$ then the rule for P has no premisses and we can conclude $P\ true$ without any inversion steps.

□

4.6 Completeness

The completeness theorem for the system with a subgoal stack states that if $A\ true$ then $A \mid \top$. It is more difficult to see how to generalize this. The following seems to work well.

Theorem 4.2 *If $A\ true$ and $\top \mid S$ then $A \mid S$.*

Proof: By induction on the structure of the deduction of $A\ true$.

Case: $\mathcal{D} = \dfrac{}{\top\ true}\ \top I$ where $A = \top$.

$\top \mid S$	Assumption
$A \mid S$	Since $A = \top$

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ A_1\ true & A_2\ true\end{array}}{A_1 \wedge A_2\ true}\ \wedge I$ where $A = A_1 \wedge A_2$.

$\top \;/\; S$	Assumption
$A_2 \;/\; S$	By ind.hyp. on $\mathcal{D}_2$
$\top \;/\; A_2 \wedge S$	By rule
$A_1 \;/\; A_2 \wedge S$	By ind.hyp. on $\mathcal{D}_1$
$A_1 \wedge A_2 \;/\; S$	By rule

Case: $\mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & & \mathcal{D}_m \\ B_1\ true & \ldots & B_m\ true\end{array}}{P\ true}$ where $A = P$.

This is similar to the previous case, except we have to repeat the pattern $m - 1$ times. One could formalize this as an auxiliary induction, but we will not bother.

$\top \;/\; S$	Assumption
$B_m \;/\; S$	By ind.hyp. on $\mathcal{D}_m$
$\top \;/\; B_m \wedge S$	By rule
$B_{m-1} \;/\; B_m \wedge S$	By ind.hyp. on $\mathcal{D}_{m-1}$
$B_{m-1} \wedge B_m \;/\; S$	By rule
$\top \;/\; (B_{m-1} \wedge B_m) \wedge S$	By rule
$B_1 \wedge \ldots \wedge B_{m-1} \wedge B_m \;/\; S$	Repeating previous 3 steps

□

This form of completeness theorem is a non-deterministic completeness theorem, since the choice which rule to apply in the case of an atomic goal remains non-deterministic. Furthermore, the instantiation of the schematic variables in the rules is "by magic": we just assume for the purpose of the semantics at this level, that all goals are ground and that the semantics will pick the right instances. We will specify how these choices are to be resolved in the next lecture.

4.7 Historical Notes

The idea of defining one language in another, similar one for the purpose of definition goes back to the early days of functional programming. The idea to transform such definitional interpreters so that advanced features are not needed in the meta-language was formulated by Reynolds [3]. After successive transformation we can arrive at an abstract machine. A very systematic account for the derivations of abstract machines in the functional

setting has been given by Danvy and several of his students (see, for example, [1]). Meta-interpreters are also common in logic programming, mostly with the goal to extend capabilities of the language. One of the earliest published account is by Bowen and Kowalski [2].

4.8 Exercises

Exercise 4.1 *Extend the meta-interpreter without goal stacks to a bounded interpreter which fails if no proof of a given depth can be found. In terms of proof trees, the depth is length of the longest path from the final conclusion to an axiom.*

Exercise 4.2 *Extend the meta-interpreter without goal stacks with loop detection, so that if while solving an atomic goal P the identical goal P arises again, that branch of the search will be terminated with failure instead of diverging. You may assume that all goals are ground.*

Exercise 4.3 *Extend the meta-interpreter with goal stacks so that if an atomic goal succeeds once, we do not search for a proof of it again but just succeed. You may assume that all goals are ground and you do not need to preserve the memo table upon backtracking.*

Exercise 4.4 *Extend the meta-interpreter with the goal stack to trace the execution of a program, printing information about the state of search.*

4.9 References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evluators and abstract machines. In *Proceedings of the 5th International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[2] Kenneth A. Bowen and Robert A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, London, 1982.

[3] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.

15-819K: Logic Programming

Lecture 5

Backtracking

Frank Pfenning

September 12, 2006

In this lecture we refine the operational semantics further to explicit represent backtracking. We prove this formulation to be sound. From earlier examples it should be clear that it can no longer be complete.

5.1 Disjunction and Falsehood

When our aim was to explicitly represent left-to-right subgoal selection, we introduced conjunction and truth as our first logical connectives. This allowed us to make the order explicit in the propositions we were interpreting.

In this lecture we would like to make the choice of rule explicit. For this purpose, it is convenient to introduce two new logical connectives: disjunction $A \vee B$ and falsehood $\bot$. They are easily defined by their introduction rules as usual.

$$\frac{A\ true}{A \vee B\ true}\ \vee I_1 \qquad \frac{B\ true}{A \vee B\ true}\ \vee I_2 \qquad \text{No } \bot I \text{ rule}$$

We can think of falsehood as a disjunction between zero alternatives; therefore, there are zero introduction rules.

5.2 Normal Forms for Programs

Sometimes it is expedient to give the semantics of programs assuming a kind of normal form. The presentation from the previous lecture would have been slightly simpler if we had presupposed an *explicit conjunction form* for programs where each inference rule has exactly one premiss. The

transformation to achieve this normal form in the presence of conjunction and truth is simple: if the rule has multiple premisses, we just conjoin them to form one premiss. If a rule has no premisses, we insert $\top$ as a premiss. It is easy to prove that this transformation preserves meaning (see Exercise 5.1).

In the semantics presented as the judgment $A \;/\; S$, every step of search is deterministic, except for selection of the rule to apply, and how to instantiate schematic rules. We will not address the latter choice in today's lecture: assume that all goals are ground, and consider for the moment only programs so that ground goals have only ground subgoals.

A simple condition on the rules that avoids any choice when encountering atomic predicates is that every ground goal matches the head of *exactly one* rule. Then, when we have a goal P the rule to apply is uniquely determined. In order to achieve this, we have to transform our program into *explicit disjunction form*. In a later lecture we will see a systematic way to create this form as part of logic program compilation. For now we are content to leave this informal and just present programs in this form.

As an example, consider the usual member predicate.

$$\frac{}{\mathsf{member}(X, [X|Ys])} \qquad \frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, [Y|Ys])}$$

There are two reasons that this predicate is not in explicit disjunction form. The first is that there is no clause for $\mathsf{member}(t, [\,])$, violating the requirement that there be exactly one clause for each ground goal. The second is that for goals $\mathsf{member}(t, [t|ys])$, both clauses apply, again violating our requirement.

We rewrite this in several steps. First, we eliminate the double occurrence of X in the first clause in favor of an explicit equality test, $X \doteq Y$.

$$\frac{X \doteq Y}{\mathsf{member}(X, [Y|Ys])} \qquad \frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, [Y|Ys])}$$

Now that the two rules have the same conclusion, we can combine them into one, using disjunction.

$$\frac{X \doteq Y \vee \mathsf{member}(X, Ys)}{\mathsf{member}(X, [Y|Ys])}$$

Finally, we add a clause for the missing case, with a premiss of falsehood.

$$\frac{\bot}{\mathsf{member}(X, [\,])} \qquad \frac{X \doteq Y \vee \mathsf{member}(X, Ys)}{\mathsf{member}(X, [Y|Ys])}$$

This program is now in explicit disjunction form, under the presupposition that the second argument to member is a list.

In Prolog, the notation for $A \vee B$ is A ; B and the notation for $\bot$ is `fail`, so the program above becomes

```
member(X, []) :- fail.
member(X, [Y | Ys]) :- X = Y ; member(X, Ys).
```

The Prolog convention is to always put whitespace around the disjunction to distinguish it more clearly from conjunction.

5.3 Equality

Transforming a program into explicit disjunction form requires equality, as we have seen in the member example. We write $s \doteq t$ for the proposition that s and t are equal, with the following introduction rule.

$$\frac{}{t \doteq t \ true}\ \doteq I$$

We will also use $s \neq t$ to denote that two terms are different as a kind of judgment.

5.4 Explicit Backtracking

Assuming the program is in explicit disjunction form, the main choice concerns how to prove $A \vee B$ as a goal. The operational semantics of Prolog prescribes that we try to solve A first and only if that fails do we try B. This suggest that in addition to the goal stack S, we add another argument F to our search judgment which records further (untried) possibilities. We refer to F as the *failure continuation* because it records what to do when the current goal A fails. We write the new judgment as $A \ / \ S \ / \ F$ and read this as: *Either A under S or F.* More formally, we will establish soundness in the form that if $A \ / \ S \ / \ F$ then $(A \wedge S) \vee F \ true$. This statement can also be our guide in designing the rules for the judgment.

First, the rules for conjunction and truth. They do not change much, just carry along the failure continuation.

$$\frac{A \ / \ B \wedge S \ / \ F}{A \wedge B \ / \ S \ / \ F} \qquad \frac{B \ / \ S \ / \ F}{\top \ / \ (B \wedge S) \ / \ F} \qquad \frac{}{\top \ / \ \top \ / \ F}$$

The rules for atomic predicates P are also simple, because we assume the given rules for truth are in explicit disjunction form.

$$\frac{B \;/\; S \;/\; F}{P \;/\; S \;/\; F} \qquad \text{for each rule} \qquad \frac{B\ \mathit{true}}{P\ \mathit{true}}$$

Next, the rule for disjunction. In analogy with conjunction and truth, it is tempting to write the following two **incorrect** rules:

$$\frac{A \;/\; S \;/\; B \vee F}{A \vee B \;/\; S \;/\; F}\ \text{incorrect} \qquad\qquad \frac{B \;/\; S \;/\; F}{\bot \;/\; S \;/\; B \vee F}\ \text{incorrect}$$

Let's try to see the case in the soundness proof for the first rule. The soundness proof proceeds by induction on the structure of the deduction for $A \;/\; S \;/\; F$. For the first of the incorrect rules we would have to show that if $(A \wedge S) \vee (B \vee F)\ \mathit{true}$ then $((A \vee B) \wedge S) \vee F\ \mathit{true}$. By inversion on the rules for disjunction, we know that either $A \wedge S\ \mathit{true}$, or $B\ \mathit{true}$, or $F\ \mathit{true}$. In the middle case ($B\ \mathit{true}$), we do not have enough information to conclude that $((A \vee B) \wedge S) \vee F\ \mathit{true}$ and the proof fails (see Exercises 5.2 and 5.3).

The failure of the soundness proof also suggests the correct rule: we have to pair up the alternative B with the goal stack S and restore S it when we backtrack to consider B.

$$\frac{A \;/\; S \;/\; (B \wedge S) \vee F}{A \vee B \;/\; S \;/\; F} \qquad\qquad \frac{B \;/\; S \;/\; F}{\bot \;/\; S' \;/\; (B \wedge S) \vee F}$$

The goal stack S' in the second rule is discarded, because it applies to the goal $\bot$ which cannot succeed. Instead we restore the goal stack S saved with B.

It is worth noting explicitly that there is one case we did not cover

$$\text{no rule for } \bot \;/\; S \;/\; \bot$$

so that our overall goal fails if the current goal fails and there are no further alternatives.

Finally, we need two rules for equality, where we appeal to the equality ($s = t$) and disequality ($s \neq t$) in our languages of judgments.

$$\frac{s = t \quad \top \;/\; S \;/\; F}{s \doteq t \;/\; S \;/\; F} \qquad\qquad \frac{s \neq t \quad \bot \;/\; S \;/\; F}{s \doteq t \;/\; S \;/\; F}$$

In a Prolog implementation this will not lead to difficulties because we assume all goals are ground, so we can always tell if two terms are equal or not.

5.5 Soundness

The soundness proof goes along familiar patterns.

Theorem 5.1 *If $A \;/\; S \;/\; F$ then $(A \wedge S) \vee F$ true.*

Proof: By induction on the structure of $\mathcal{D}$ of $A \;/\; S \;/\; F$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ A_1 \;/\; A_2 \wedge S \;/\; F\end{array}}{A_1 \wedge A_2 \;/\; S \;/\; F}$ where $A = A_1 \wedge A_2$.

$(A_1 \wedge (A_2 \wedge S)) \vee F$ *true*	By ind.hyp. on $\mathcal{D}_1$
$A_1 \wedge (A_2 \wedge S)$ *true* or F *true*	By inversion
$A_1 \wedge (A_2 \wedge S)$ *true*	First subcase
A_1 *true* and A_2 *true* and S *true*	By two inversions
$(A_1 \wedge A_2) \wedge S$ *true*	By two rule applications
$((A_1 \wedge A_2) \wedge S) \vee F$ *true*	By rule $(\vee I_1)$
F *true*	Second subcase
$((A_1 \wedge A_2) \wedge S) \vee F$ *true*	By rule $(\vee I_2)$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2 \\ A_2 \;/\; S_1 \;/\; F\end{array}}{\top \;/\; (A_2 \wedge S_1) \;/\; F}$ where $A = \top$ and $S = A_2 \wedge S_1$. Similar to the previous case.

Case: $\mathcal{D} = \dfrac{}{\top \;/\; \top \;/\; F}$ where $A = S = \top$.

$\top$ *true*	By rule $(\top I)$
$\top \wedge \top$ *true*	By rule $(\wedge I)$
$(\top \wedge \top) \vee F$ *true*	By rule $\vee I_1$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ B \;/\; S \;/\; F\end{array}}{P \;/\; S \;/\; F}$ where $A = P$ and $\dfrac{B \text{ true}}{P \text{ true}}$.

$(B \wedge S) \vee F$ *true*	By ind.hyp. on $\mathcal{D}'$
B *true* and S *true*	First subcase, after inversion

$P\ true$	By rule
$(P \wedge S) \vee F$	By rules ($\wedge I$ and $\vee I_1$)
$F\ true$	Second subcase, after inversion
$(P \wedge S) \vee F$	By rule ($\vee I_2$)

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ A_1 \ / \ S \ / \ (A_2 \wedge S) \vee F\end{array}}{A_1 \vee A_2 \ / \ S \ / \ F}$ where $A = A_1 \vee A_2$.

$(A_1 \wedge S) \vee ((A_2 \wedge S) \vee F)\ true$	By ind.hyp. on $\mathcal{D}_1$
$A_1\ true$ and $S\ true$	First subcase, after inversion
$A_1 \vee A_2\ true$	By rule ($\vee I_1$)
$((A_1 \vee A_2) \wedge S) \vee F\ true$	By rules ($\wedge I$ and $\vee I_1$)
$A_2\ true$ and $S\ true$	Second subcase, after inversion
$A_1 \vee A_2\ true$	By rule ($\vee I_2$)
$((A_1 \vee A_2) \wedge S) \vee F\ true$	By rules ($\wedge I$ and $\vee I_1$)
$F\ true$	Third subcase, after inversion
$((A_1 \vee A_2) \wedge S) \vee F\ true$	By rule ($\vee I_2$)

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_2 \\ A_2 \ / \ S_1 \ / \ F_0\end{array}}{\bot \ / \ S \ / \ (A_2 \wedge S_1) \vee F_0}$ where $A = \bot$ and $F = (A_2 \wedge S_1) \vee F_0$.

$(A_2 \wedge S_1) \vee F_0\ true$	By ind.hyp. on $\mathcal{D}_2$
$(\bot \wedge S) \vee ((A_2 \wedge S_1) \vee F_0)\ true$	By rule ($\vee I_2$)

Cases: The cases for equality are left to the reader (see Exercise 5.4).

□

5.6 Completeness

Of course, the given set of rules is *not* complete. For example, the single rule

$$\frac{\mathsf{diverge} \vee \top}{\mathsf{diverge}}$$

cannot be found by search, that is, there is no proof of

$$\mathsf{diverge} \;/\; \top \;/\; \bot$$

even though there is a simple proof that diverge $true$.

$$\dfrac{\dfrac{}{\top\ true}\ \top I}{\dfrac{}{\mathsf{diverge} \vee \top\ true}\ \vee I_2}{\mathsf{diverge}\ true}$$

However, it is interesting to consider that the set of rules is complete a weaker sense, namely that if $A \;/\; \top \;/\; \bot$ can be reduced to $\bot \;/\; S \;/\; \bot$ then there can be no proof of $A\ true$. We will not do this here (see Exercise 5.5). One way to approach this formally is to add another argument and use a four-place judgment

$$A \;/\; S \;/\; F \;/\; J$$

where J is either istrue (if a proof can be found) or isfalse (if the the attempt to find a proof fails finitely).

5.7 A Meta-Interpreter with Explicit Backtracking

Based on the idea at the end of the last section, we can turn the inference system into a Prolog program that can tell us explicitly whenever search succeeds or fails finitely.

Recall the important assumption that all goals are ground, and that the program is in explicit disjunction form.

```
solve(true, true, _, istrue).
solve(true, (A , S), F, J) :- solve(A, S, F, J).
solve((A , B), S, F, J) :- solve(A, (B, S), F, J).
solve(fail, _, fail, isfalse).
solve(fail, _, ((B , S) ; F), J) :- solve(B, S, F, J).
solve((A ; B), S, F, J) :- solve(A, S, ((B , S) ; F), J).
solve((X = Y), S, F, J) :- X = Y, solve(true, S, F, J).
solve((X = Y), S, F, J) :- X \= Y, solve(fail, S, F, J).
solve(P, S, F, J) :- clause(P, B), solve(B, S, F, J).

% top level interface
solve(A, J) :- solve(A, true, fail, J).
```

Given a program in explicit disjunction form, such as

```
member(X, []) :- fail.
member(X, [Y|Ys]) :- X = Y ; member(X, Ys).
```

we can now ask

```
?- solve(member(1, [2,3,4]), J).
J = isfalse;

?- solve(member(1, [1,2,1,4]), J).
J = istrue;
```

Each query will succeed only once since our meta-interpreter only searches for the first solution (see Exercise 5.6).

5.8 Abstract Machines

The meta-interpreter in which both subgoal selection and backtracking are explicit comes close to the specification of an abstract machine. In order to see how the inference rule can be seen as transition rules, we consider $A \mid S \mid F$ as the state of the machine. Each rule for this judgment has only one premiss, so each rule, when read from the conclusion to the premiss can be seen as a transition rule for an abstract machine.

Examining the rules we can see that for every state there is a unique state transition, with the following exceptions:

1. A state $\top \mid \top \mid F$ is final since there is no premiss for the maching rule. The computation finishes.

2. A state $\bot \mid S \mid \bot$ is final since there is no rule that applies. The computation fails.

3. A state $P \mid S \mid F$ applies a unique transition (by the requirement that there be a unique rule for every atomic goal P), although how to find that rule instance remains informal.

In the next lecture we make the process of rule application more precise, and we also admit goals with free variables as in Prolog.

5.9 Historical Notes

The explicit disjunction form is a pre-cursor of Clark's *iff-completion* of a program [2]. This idea is quite general, is useful in compilation of logic programs, and can be applied to much richer logic programming languages than Horn logic [1].

Early logic programming theory generally did not make backtracking explicit. Some references will appear in the next lecture, since some of the intrinsic interest arises from the notion of substitution and unification.

5.10 Exercises

Exercise 5.1 *Prove that if we replace every rule*

$$\frac{B_1\ true \quad \ldots \quad B_m\ true}{P\ true}$$

by

$$\frac{B_1 \wedge \cdots \wedge B_m\ true}{P\ true}$$

to achieve the explicit conjunction form*, the original and revised specification are strongly equivalent in the sense that there is a bijection between the proofs in the two formulations for each atomic proposition. Read the empty conjunction ($m = 0$) as $\top$.*

Exercise 5.2 *Give a counterexample to show that the failure in the soundness proof for the first incorrect disjunction rule is not just a failure in the proof: the system is actually unsound with respect to logical truth.*

Exercise 5.3 *Investigate if the second incorrect rule for disjunction*

$$\frac{B\ /\ S\ /\ F}{\bot\ /\ S\ /\ B \vee F}\ \textit{incorrect}$$

also leads to a failure in the soundness proof. If so, give a counterexample. If not, discuss in what sense this rule is nonetheless incorrect.

Exercise 5.4 *Complete the soundness proof by giving the cases for equality.*

Exercise 5.5 *Investigate weaker notions of completeness for the backtracking semantics, as mentioned at the end of the section of completeness.*

Exercise 5.6 *Rewrite the meta-interpreter so it counts the number of proofs of a goal instead of returning just an indication of whether it is true or false. You may make all the same assumptions that* `solve(`A`, `S`, `F`, `J`)` *makes.*

Exercise 5.7 *Revisit the example from Lecture 3*

$$\frac{}{\mathsf{digit}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))))))} \qquad \frac{\mathsf{digit}(\mathsf{s}(N))}{\mathsf{digit}(N)}$$

and prove more formally now that any query `?- digit(`n`)` *for* $n > 9$ *will not terminate. You should begin by rewriting the program into explicit disjunction form. Please be clear what you are doing the induction over (if you use induction), and explain in which way your theorem captures the statement above.*

Exercise 5.8 *If we were not concerned about space usage or efficiency, we could write a breadth-first interpreter instead of backtracking. Specify such an interpreter using the judgmental style and prove that it is sound and complete with respect to logical truth. Translate your interpreter into a Prolog program.*

5.11 References

[1] Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 115–129, Manchester, England, June 1998. MIT Press.

[2] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

15-819K: Logic Programming

Lecture 6

Unification

Frank Pfenning

September 14, 2006

In this lecture we take the essential step towards making the choice of goal and rule instantiation explicit in the operational semantics. This consists of describing an algorithm for a problem called *unification* which, given two terms t and s, tries to find a substitution θ for its free variables such that $t\theta = s\theta$ if such a substitution exists. Recall that we write $t\theta$ for the result of applying the substitution θ to the term t.

6.1 Using Unification in Proof Search

Before we get to specifics of the algorithm, we consider how we use unification in proof search. Let us reconsider the (by now tired) example of unary addition

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad\qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

and an atomic goal such as

$$\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), P).$$

Clearly the conclusion of the first rule does not match this goal, but the second one does. What question do we answer to arrive at this statement?

The first attempt might be: "*There is an instance of the rule such that the conclusion matches the goal.*" When we say *instance* we mean here the result of substituting terms for the variables occuring in a rule, proposition, or term. We can see that this specification is not quite right: we need to instantiate the goal as well, since P must have the form $\mathsf{s}(P_1)$ for some as yet

unknown P_1. The subgoal in that case would be $\mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})), P_1)$ according to the rule instantiation with z for M, $\mathsf{s}(\mathsf{s}(\mathsf{z}))$ for N, and P_1 for P.

The second attempt would therefore be: "*There is an instance of the rule and an instance of the goal such that the two are equal.*" This does not quite capture what we need either. For example, substituting $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))$ for P in the goal and $\mathsf{s}(\mathsf{s}(\mathsf{z})))$ for P in the rule, together with the substitution for M and N from above, will also make the goal and conclusion of the rule identical, but is nonetheless wrong. The problem is that it would overcommit: using P_1 for P in the rule, on the other hand, keeps the options open. P_1 will be determined later by search and other unifications. In order to express this, we define that t_1 is *more general* than t_2 if t_1 can be instantiated to t_2.

The third attempt is therefore: "*Find the most general instance of the rule and the goal so that the conclusion of the rule is equal to the instantiated goal.*" Phrased in terms of substitutions, this says: find θ_1 and θ_2 such that $P'\theta_1 = P\theta_2$, and any other common instance of P' and P is an instance of $P'\theta_1$.

In terms of the algorithm description it is more convenient if we redefine the problem slightly in this way: "*First rename the variables in the rule so that they are disjoint from the variables in the goal. Then find a single most general substitution θ that unifies the renamed conclusion with the goal.*" Here, a unifying substitution θ is most general if any other unifying substitution is an instance of θ.

In the remainder of the lecture we will make these notions more precise and present an algorithm to compute a most general unifier. In the next lecture we show how to reformulate the operational semantics to explicitly use most general unifiers. This means that for the first time the semantics will admit free variables in goals.

6.2 Substitutions

We begin with a specification of substitutions. We use the notation $\mathrm{FV}(t)$ for the set of all free variables in a term.

$$\text{Substitutions} \quad \theta \quad ::= \quad t_1/x_1, \ldots, t_n/x_n$$

We postulate that all x_i are distinct. The order of the pairs in the substitution is irrelevant, and we consider permutations of substitutions to be equal. We denote the *domain* of θ by $\mathrm{dom}(\theta) = \{x_1, \ldots, x_n\}$. Similarly, we call the set of all variables occurring in the substitution term t_i the *codomain* and write $\mathrm{cod}(\theta) = \bigcup_i \mathrm{FV}(t_i)$.

An important general assumption is that the domain and codomain of substitutions are disjoint.

> **Assumption:** All substitutions we consider are *valid*, that is, $\mathrm{dom}(\theta) \cap \mathrm{cod}(\theta) = \emptyset$ for any substitution θ.

This is by no means the only way to proceed. A more common assumption is that all substitutions are idempotent, but we believe the above is slightly more convenient for our limited purposes.

Note that valid substitutions cannot contain pairs x/x, since it would violate our assumption above. However, such pairs are not needed, since their action is the identity, which can also be achieve by simply omitting them.

Applying a substitution θ to a term t is easily defined compositionally.

$$\begin{array}{lcll} x\theta & = & t & \text{if } t/x \text{ in } \theta \\ y\theta & = & y & \text{if } y \notin \mathrm{dom}(\theta) \\ f(t_1, \ldots, t_n)\theta & = & f(t_1\theta, \ldots, t_n\theta) & \end{array}$$

6.3 Composing Substitutions

In the course of search for a deduction, and even in the course of solving one unification problem, we obtain information in a piecemeal fashion. This means we construct a (partial) substitution, apply it, and then construct another substitution on the result. The overall answer is then the *composition* of these two substitution. We write it as $\tau\theta$. The guiding property we need is that for any t, we have $(t\tau)\theta = t(\tau\theta)$. In order for this property to hold and maintain our general assumptions on substitutions, we specify the precondition that $\mathrm{dom}(\tau) \cap \mathrm{dom}(\theta) = \emptyset$ and also that $\mathrm{dom}(\tau) \cap \mathrm{cod}(\theta) = \emptyset$. We define the composition by going through the substitution left-to-right, until encountering the empty substitution $(\cdot)$.

$$\begin{array}{lcl} (t/x, \tau)\theta & = & t\theta/x, \tau\theta \\ (\cdot)\theta & = & \theta \end{array}$$

First, note that $\mathrm{dom}(\tau\theta) = \mathrm{dom}(\tau) \cup \mathrm{dom}(\theta)$ which is a union of disjoint domains. Second, $\mathrm{cod}(\tau\theta) = (\mathrm{cod}(\tau) - \mathrm{dom}(\theta)) \cup \mathrm{cod}(\theta)$ so that $\mathrm{cod}(\tau\theta) \cap \mathrm{dom}(\tau\theta) = \emptyset$ as can be seen by calculation. In other words, $\tau\theta$ is a valid substitution.

It is easy to verify that the desired property of composition actually holds. We will also need a corresponding property stating that composition of substitution is associative, under suitable assumptions.

Theorem 6.1 (Substitution Composition) *Assume we are given a term t and valid substitutions σ and θ with $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\theta) = \emptyset$ and $\mathrm{dom}(\sigma) \cap \mathrm{cod}(\theta) = \emptyset$. Then $\sigma\theta$ is valid and*

$$(t\sigma)\theta = t(\sigma\theta)$$

Furthermore, if τ is a substitution such that $\mathrm{dom}(\tau) \cap \mathrm{dom}(\sigma) = \mathrm{dom}(\tau) \cap \mathrm{dom}(\theta) = \emptyset$ then also

$$(\tau\sigma)\theta = \tau(\sigma\theta)$$

Proof: The validity of $\sigma\theta$ has already been observed above. The first equality follows by induction on the structure of the term t, the second by induction on the structure of τ (see Exercise 6.1).

Case: $t = x$ for a variable x. Then we distinguish two subcases.

Subcase: $x \in \mathrm{dom}(\sigma)$ where $s/x \in \sigma$.

$$\begin{array}{ll} (x\sigma)\theta = s\theta & \text{By defn. of } x\sigma \\ = x(\sigma\theta) & \text{Since } s\theta/x \in \sigma\theta \end{array}$$

Subcase: $x \notin \mathrm{dom}(\sigma)$.

$$\begin{array}{ll} (x\sigma)\theta = x\theta & \text{By defn. of } x\sigma \\ = x(\sigma\theta) & \text{By defn. of } \sigma\theta \end{array}$$

Case: $t = f(t_1, \ldots, t_n)$ for terms $t_1, \ldots, t_n$.

$$\begin{array}{ll} (t\sigma)\theta = f(t_1\sigma, \ldots, t_n\sigma)\theta & \text{By defn. of } t\sigma \\ = f((t_1\sigma)\theta, \ldots, (t_n\sigma)\theta) & \text{By defn. of } f(\_)\theta \\ = f(t_1(\sigma\theta), \ldots, t_n(\sigma\theta)) & \text{By i.h., } n \text{ times} \\ = f(t_1, \ldots t_n)(\sigma\theta) = t(\sigma\theta) & \text{By defn. of } t(\sigma\theta) \end{array}$$

□

6.4 Unification

We say θ is a *unifier* of t and s if $t\theta = s\theta$. We say that θ is a *most general unifier* for t and s if it is a unifier, and for any other unifier σ there exists a substitution σ' such that $\sigma = \theta\sigma'$. In other words, a unifier is most general if any other unifier is an instance of it, where "instance" refers to the composition of substitutions.

As usual in this class, we present the algorithm to compute a most general unifier as a judgment, via a set of inference rules. The judgment has

the form $t \doteq s \mid \theta$, where we think of t and s as inputs and a most general unifier θ as the output. In order to avoid the n-ary nature of the list of arguments, we will have an auxiliary judgment $\mathbf{t} \doteq \mathbf{s} \mid \theta$ for sequences of terms $\mathbf{t}$ and $\mathbf{s}$. Notions such as application of substitution are extended to sequences of terms in the obvious way. We use $(\cdot)$ to stand for an empty sequence of terms (as well as the empty substitution, which is a sequence of term and variable pairs). In general, we will use boldface letters to stand for sequences of terms.

We first consider function terms and term sequences.

$$\frac{\mathbf{t} \doteq \mathbf{s} \mid \theta}{f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta} \qquad \frac{}{(\cdot) \doteq (\cdot) \mid (\cdot)} \qquad \frac{t \doteq s \mid \theta_1 \quad \mathbf{t}\theta_1 \doteq \mathbf{s}\theta_1 \mid \theta_2}{(t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid \theta_1\theta_2}$$

Second, the cases for variables.

$$\frac{}{x \doteq x \mid (\cdot)} \qquad \frac{x \notin \mathrm{FV}(t)}{x \doteq t \mid (t/x)} \qquad \frac{t = f(\mathbf{t}), x \notin \mathrm{FV}(t)}{t \doteq x \mid (t/x)}$$

The condition that $t = f(\mathbf{t})$ in the last rule ensures that it does not overlap with the rule for $x \doteq t$. The condition that $x \notin \mathrm{FV}(t)$ is necessary because, for example, the two terms x and $f(x)$ do not have unifier: no matter what, the substitution $f(x)\theta$ will always have one more occurrence of f than $x\theta$ and hence the two cannot be equal.

The other situations where unification fails is an equation of the form $f(\mathbf{t}) = g(\mathbf{s})$ for $f \neq g$, and two sequences of terms of unequal length. The latter can happen if function symbols are overloaded at different arities, in which case failure of unification is the correct result.

6.5 Soundness

There are a number of properties we would like to investigate regarding the unification algorithm proposed in the previous section. The first is its soundness, that is, we would like to show that the substitution θ is indeed a unifier.

Theorem 6.2 *If $t \doteq s \mid \theta$ then $t\theta = s\theta$.*

Proof: We need to generalize this to cover the auxiliary unification judgment on term sequences.

(i) If $t \doteq s \mid \theta$ then $t\theta = s\theta$.

(ii) If $\mathbf{t} \doteq \mathbf{s} \mid \theta$ then $\mathbf{t}\theta = \mathbf{s}\theta$.

The proof proceeds by mutual induction on the structure of the deduction $\mathcal{D}$ of $t \doteq s$ and $\mathcal{E}$ of $\mathbf{t} \doteq \mathbf{s}$. This means that if one judgment appears as in the premiss of a rule for the other, we can apply the appropriate induction hypothesis.

In the proof below we will occasionally refer to *equality reasoning*, which refers to properties of equality in our mathematical language of discourse, not properties of the judgment $t \doteq s$. There are also some straightforward lemmas we do not bother to prove formally, such as $t(s/x) = t$ if $x \notin \mathrm{FV}(t)$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{E}\\ \mathbf{t} \doteq \mathbf{s} \mid \theta\end{array}}{f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta}$ where $t = f(\mathbf{t})$ and $s = f(\mathbf{s})$.

$\mathbf{t}\theta = \mathbf{s}\theta$ By i.h.(ii) on $\mathcal{E}$
$f(\mathbf{t})\theta = f(\mathbf{s})\theta$ By definition of substitution

Case: $\mathcal{D} = \dfrac{}{(\cdot) \doteq (\cdot) \mid (\cdot)}$ where $\mathbf{t} = \mathbf{s} = (\cdot)$ and $\theta = (\cdot)$.

$(\cdot)\theta = (\cdot)\theta$ By equality reasoning

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}_2\\ t_1 \doteq s_1 \mid \theta_1 & \mathbf{t}_2\theta_1 \doteq \mathbf{s}_2\theta_1 \mid \theta_2\end{array}}{(t_1, \mathbf{t}_2) \doteq (s_1, \mathbf{s}_2) \mid \theta_1\theta_2}$ where $\mathbf{t} = (t_1, \mathbf{t}_2)$ and $\mathbf{s} = (s_1, \mathbf{s}_2)$ and $\theta = \theta_1\theta_2$.

$t_1\theta_1 = s_1\theta_1$ By i.h.(i) on $\mathcal{D}_1$
$(t_1\theta_1)\theta_2 = (s_1\theta_1)\theta_2$ By equality reasoning
$t_1(\theta_1\theta_2) = s_1(\theta_2\theta_2)$ By substitution composition (Theorem 6.1)
$(\mathbf{t}_2\theta_1)\theta_2 = (\mathbf{s}_2\theta_1)\theta_2$ By i.h.(ii) on $\mathcal{E}_2$
$\mathbf{t}_2(\theta_1\theta_2) = \mathbf{s}_2(\theta_1\theta_2)$ By substitution composition
$(t_1, \mathbf{t}_2)(\theta_1\theta_2) = (s_1, \mathbf{s}_2)(\theta_1\theta_2)$ By defn. of substitution

Case: $\mathcal{D} = \dfrac{}{x \doteq x \mid (\cdot)}$ where $t = s = x$ and $\theta = (\cdot)$.

$x(\cdot) = x(\cdot)$ By equality reasoning

Case: $\mathcal{D} = \dfrac{x \notin \mathrm{FV}(s)}{x \doteq s \mid (s/x)}$ where $t = x$ and $\theta = (s/x)$.

$$\begin{array}{ll} x(s/x) = s & \text{By defn. of substitution} \\ = s(s/x) & \text{Since } x \notin \mathrm{FV}(s) \end{array}$$

Case: $\mathcal{D} = \dfrac{t = f(\mathbf{t}), x \notin \mathrm{FV}(t)}{t \doteq x \mid (t/x)}$ where $s = x$ and $\theta = (t/x)$.

$$\begin{array}{ll} t(t/x) = t & \text{Since } x \notin \mathrm{FV}(t) \\ = x(t/x) & \text{By defn. of substitution} \end{array}$$

□

6.6 Completeness

Completness of the algorithm states that if s and t have a unifer then there exists a most general one according to the algorithm. We then also need to observe that the unification judgment is deterministic to see that, if interpreted as an algorithm, it will always find a most general unifier if one exists.

Theorem 6.3 *If $t\sigma = s\sigma$ then $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.*

Proof: As in the soundness proof, we generalize to address sequences.

(i) If $t\sigma = s\sigma$ then $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$.

(ii) If $\mathbf{t}\sigma = \mathbf{s}\sigma$ then $\mathbf{t} \doteq \mathbf{s} \mid \theta$ and $\sigma = \theta\sigma'$.

The proof proceeds by mutual induction on the structure of $t\sigma$ and $\mathbf{t}\sigma$. We proceed by distinguishing cases for t and s, as well as $\mathbf{t}$ and $\mathbf{s}$. This structure of argument is a bit unusual: mostly, we distinguish cases of the subject of our induction, be it a deduction or a syntactic object. In the situation here it is easy to make a mistake and incorrectly attempt to apply the induction hypothesis, so you should carefully examine all appeals to the induction hypothesis below to make sure you understand why they are correct.

Case: $t = f(\mathbf{t})$. In this case we distinguish subcases for s.

Subcase: $s = f(\mathbf{s})$.

$f(\mathbf{t})\sigma = f(\mathbf{s})\sigma$	Assumption
$\mathbf{t}\sigma = \mathbf{s}\sigma$	By defn. of substitution
$\mathbf{t} \doteq \mathbf{s} \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'	By i.h.(ii) on $\mathbf{t}\sigma$
$f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta$	By rule

Subcase: $s = g(\mathbf{s})$ for $f \neq g$. This subcase is impossible:

$f(\mathbf{t})\sigma = g(\mathbf{s})\sigma$	Assumption
Contradiction	By defn. of substitution

Subcase: $s = x$.

$f(\mathbf{t})\sigma = x\sigma$	Assumption
$\sigma = (f(\mathbf{t})\sigma/x, \sigma')$ for some σ'	By defn. of subst. and reordering
$x \notin \mathrm{FV}(f(\mathbf{t}))$	Otherwise $f(\mathbf{t})\sigma \neq x\sigma$
$f(\mathbf{t}) = x \mid (f(\mathbf{t})/x)$ so we let $\theta = (f(\mathbf{t})/x)$	By rule
$\sigma = (f(\mathbf{t})\sigma/x, \sigma')$	See above
$= (f(\mathbf{t})\sigma'/x, \sigma')$	Since $x \notin \mathrm{FV}(f(\mathbf{t}))$
$= (f(\mathbf{t})/x)\sigma'$	By defn. of substitution
$= \theta\sigma'$	Since $\theta = (f(\mathbf{t})/x)$

Case: $t = x$. In this case we also distinguish subcases for s and proceed symmetrically to the above.

Case: $\mathbf{t} = (\cdot)$. In this case we distinguish cases for $\mathbf{s}$.

Subcase: $\mathbf{s} = (\cdot)$.

$(\cdot) \doteq (\cdot) \mid (\cdot)$	By rule
$\sigma = (\cdot)\sigma$	By defn. of substitution

Subcase: $\mathbf{s} = (s_1, \mathbf{s}_2)$. This case is impossible:

$(\cdot)\sigma = (s_1, \mathbf{s}_2)\sigma$	Assumption
Contradiction	By definition of substitution

Case: $\mathbf{t} = (t_1, \mathbf{t}_2)$. Again, we distinguish two subcases.

Subcase: $\mathbf{s} = (\cdot)$. This case is impossible, like the symmetric case above.

Subcase: $\mathbf{s} = (s_1, \mathbf{s}_2)$.

$(t_1, \mathbf{t}_2)\sigma = (s_1, \mathbf{s}_2)\sigma$	Assumption
$t_1\sigma = s_1\sigma$ and	
$\mathbf{t}_2\sigma = \mathbf{s}_2\sigma$	By defn. of substitution

$t_1 \doteq s_1 \mid \theta_1$ and
$\sigma = \theta_1\sigma_1'$ for some θ_1 and σ_1' By i.h.(i) on $t_1\sigma$
$\mathbf{t}_2(\theta_1\sigma_1') = \mathbf{s}_2(\theta_1\sigma_1')$ By equality reasoning
$(\mathbf{t}_2\theta_1)\sigma_1' = (\mathbf{s}_2\theta_1)\sigma_1'$ By subst. composition (Theorem 6.1)
$\mathbf{t}_2\theta_1 \doteq \mathbf{s}_2\theta_1 \mid \theta_2$ and
$\sigma_1' = \theta_2\sigma_2'$ for some θ_2 and σ_2' By i.h.(ii) on $\mathbf{t}_2\sigma$ $(= (\mathbf{t}_2\theta_1)\sigma_1')$
$(t_1, \mathbf{t}_2) \doteq (s_1, \mathbf{s}_2) \mid \theta_1\theta_2$ By rule
$\sigma = \theta_1\sigma_1' = \theta_1(\theta_2\sigma_2')$ By equality reasoning
$= (\theta_1\theta_2)\sigma_2'$ By subsitution composition (Theorem 6.1)

□

It is worth observing that a proof by mutual induction on the structure of t and $\mathbf{t}$ would fail here (see Exercise 6.2).

An alternative way we can state the first induction hypothesis is:

> For all r, s, t, and σ such that $r = t\sigma = s\sigma$, there exists a θ and a σ' such that $t \doteq s \mid \theta$ and $\sigma = \theta\sigma'$.

The the proof is by induction on the structure of r, although the case we distinguish still concern the structure of s and t.

6.7 Termination

From the completeness argument in the previous section we can see that if given t and s the deduction of $t \doteq s \mid \theta$ is bounded by the structure of the common instance $r = t\theta = s\theta$. Since the rules furthermore have no non-determinism and the occurs-checks in the variable/term and term/variable cases also just traverse subterms of r, it means a unifier (if it exists) can be found in time proportional to the size of r.

Unfortunately, this means that this unification algorithm is exponential in the size of t and s. For example, the only unifier for

$$g(x_0, x_1, x_2, \ldots, x_n) \doteq g(f(x_1, x_1), f(x_2, x_2), f(x_3, x_3), \ldots a)$$

has 2^n occurrences of a.

Nevertheless, it is this exponential algorithm with a small, but significant modification that is used in Prolog implementations. This modification (which make Prolog unsound from the logical perspective!) is to omit the check $x \notin \mathrm{FV}(t)$ in the variable/term and term/variable cases and construct a circular term. This means that the variable/term case in unification

is constant time, because in an implementation we just change a pointer associated with the variable to point to the term. This is of crucial importance, since unification in Prolog models parameter-passing from other languages (thinking of the predicate as a procedure), and it is not acceptable to take time proportional to the size of the argument to invoke a procedure.

This observation notwithstanding, the worst-case complexity of the algorithm in Prolog is still exponential in the size of the input terms, but it is linear in the size of the result of unification. The latter fact appears to be what rescues this algorithm in practice, together with its straightforward behavior which is important for Prolog programmers.

All of this does not tell us what happens if we pass terms to our unification algorithm that do *not* have a unifier. It is not even obvious that the given rules terminate in that case (see Exercise 6.3). Fortunately, in practice most non-unifiable terms result in a clash between function symbols rather quickly.

6.8 Historical Notes

Unification was originally developed by Robinson [7] together with resolution as a proof search principle. Both of these critically influenced the early designs of Prolog, the first logic programming language. Similar computations were described before, but not studied in their own right (see [1] for more on the history of unification).

It is possible to improve the complexity of unification to linear in the size of the input terms if a different representation for the terms and substitutions is chosen, such as a set of multi-equations [4, 5] or dag structures with parent pointers [6]. These and similar algorithms are important in some applications [3], although in logic programming and general theorem proving, minor variants of Robinson's original algorithm are prevalent.

Most modern versions of Prolog support sound unification, either as a separate predicate `unify_with_occurs_check/2` or even as an optional part of the basic execution mechanism[1]. Given advanced compilation technology, I have been quoted figures of 10% to 15% overhead for using sound unification, but I have not found a definitive study confirming this. We will return to the necessary optimization in a later lecture.

Another way out is to declare that the bug is a feature, and Prolog is really a constraint programming language over rational trees, which requires a small modification of the unification algorithm to ensure termination in

[1] for example, in Amzi!Prolog

the presence of circular terms [2] but still avoids the occurs-check. The price to be paid is that the connection to the predicate calculus is lost, and that popular reasoning techniques such as induction are much more difficult to apply in the presence of infinite terms.

6.9 Exercises

Exercise 6.1 *Prove $\tau(\sigma\theta) = (\tau\sigma)\theta$ under the conditions stated in Theorem 6.1.*

Exercise 6.2 *Show precisely where and why the attempt to prove completeness of the rules for unification by mutual induction over the structure of t and $\mathbf{t}$ (instead of $t\sigma$ and $\mathbf{t}\sigma$) would fail.*

Exercise 6.3 *Show that the rules for unification terminate no matter whether given unifiable or non-unifiable terms t and s. Together with soundness, completeness, and determinacy of the rules this means that they constitute a decision procedure for finding a most general unifier if it exists.*

6.10 References

[1] Franz Baader and Wayne Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 8, pages 447–532. Elsevier and MIT Press, 2001.

[2] Joxan Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2(3):207–219, 1984.

[3] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[4] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[5] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[6] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[7] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

15-819K: Logic Programming

Lecture 7

Lifting

Frank Pfenning

September 19, 2006

Lifting is the name for turning a search calculus with ground judgments into one employing free variables. Unification might be viewed as the result of lifting a ground equality judgment, but we never explicitly introduced such a judgment. In this lecture our goal is to lift previously given operational semantics judgments for logic programming to permit free variables and prove their soundness and completeness. Unification and the judgment to compute most general unifiers are the central tools.

7.1 Explicating Rule Application

When explicating the left-to-right subgoal selection order in the operational semantics, we needed to introduce conjunction and truth in order to expose these choices explicitly. When explicating the first-to-last strategy of clause selection and backtracking we need to introduce disjunction and falsehood. It should therefore come as no surprise that in order to explicit the details of rule application and unification we need some further logical connectives. These include at least universal quantification and implication. However, we will for the moment postpone their formal treatment in order to concentrate on the integration of unification from the previous lecture into the operational semantics.

An inference rule

$$\frac{B\ true}{P\ true}$$

is modeled logically as the proposition

$$\forall x_1 \ldots \forall x_n.\, B \supset P$$

where $\{x_1, \ldots, x_n\}$ is the set of free variables in the rule. A logic program is a collection of such propositions. According to the meaning of the universal quantifier, we can use such a proposition by instantiating the universally quantified variables with arbitrary terms. If we denote this substitution by τ with $\mathrm{dom}(\tau) = \{x_1, \ldots, x_n\}$ and $\mathrm{cod}(\tau) = \emptyset$, then instantiating the quantifiers yields $B\tau \supset P\tau\ \mathit{true}$. We can use this implication to conclude $P\tau\ \mathit{true}$ if we have a proof of $B\tau\ \mathit{true}$. Require the co-domain of τ to be empty means that the substitution is ground: there are no variables in its substitution terms. This is to correctly represent the convention that an inference rule stands for all of its ground instances.

In order to match Prolog syntax more closely, we often write $P \leftarrow B$ for $B \supset P$. Moreover, we abbreviate a whole sequence of quantifiers as $\forall \mathbf{x}.\, A$, use $\mathbf{x}$ for a set of variables.

Previously, for every rule

$$\frac{B\ \mathit{true}}{P\ \mathit{true}}$$

we add a rule

$$\frac{B\ /\ S}{P\ /\ S}$$

to the operational semantics judgment $A\ /\ S$.

Now we want to replace all these rules by a single rule. This means we have to make the program explicit as a collection Γ of propositions, representing the rules as propositions. We always assume that all members of Γ are closed, that is, $\mathrm{FV}(A) = \emptyset$ for all $A \in \Gamma$. We write this judgment as

$$\Gamma \vdash A\ /\ S$$

which means that A under stack S follows from program Γ. Rule application then has the form

$$\frac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \qquad \begin{array}{l}\mathrm{dom}(\tau) = \mathbf{x} \\ \mathrm{cod}(\tau) = \emptyset \\ P'\tau = P\end{array} \qquad \Gamma \vdash B'\tau\ /\ S}{\Gamma \vdash P\ /\ S}$$

All the other rules just carry the program Γ along, since it remains fixed. We will therefore suppress it when writing the judgment.

7.2 Free Variable Deduction

We use the calculus with an explicit goal stack as the starting point. We recall the rules, omitting $\Gamma \vdash$ as promised.

$$\frac{A \mathbin{/} B \wedge S}{A \wedge B \mathbin{/} S} \qquad \frac{B \mathbin{/} S}{\top \mathbin{/} B \wedge S} \qquad \frac{}{\top \mathbin{/} \top}$$

$$\frac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \quad \begin{array}{l}\mathrm{dom}(\tau) = \mathbf{x} \\ \mathrm{cod}(\tau) = \emptyset \\ P'\tau = P\end{array} \qquad B'\tau \mathbin{/} S}{P \mathbin{/} S}$$

In the free variable form we return a substitution θ, which is reminiscent of our formulation of unification. The rough idea, formalized in the next section, is that if $A \mathbin{/} S \mid \theta$ then $A\theta \mathbin{/} S\theta$. The first three rules are easily transformed.

$$\frac{A \mathbin{/} B \wedge S \mid \theta}{A \wedge B \mathbin{/} S \mid \theta} \qquad \frac{B \mathbin{/} S \mid \theta}{\top \mathbin{/} B \wedge S \mid \theta} \qquad \frac{}{\top \mathbin{/} \top \mid (\cdot)}$$

The rule for atomic goals requires a bit of thought. In order to avoid a conflict between the names of the variables in the rule, and the names of variables in the goal, we apply a so-called *renaming substitution*. A renaming substitution ρ has the form $y_1/x_1, \ldots, y_n/x_n$ where all the x_i and y_i are distinct. We will always use ρ to denote renaming substitutions. In Prolog terminology we say that we *copy the clause*, instantiating its variables with fresh variables.

$$\frac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \quad \begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ \mathrm{cod}(\rho) \cap \mathrm{FV}(P/S) = \emptyset \\ P'\rho \doteq P \mid \theta_1\end{array} \qquad B'\rho\theta_1 \mathbin{/} S\theta_1 \mid \theta_2}{P \mathbin{/} S \mid \theta_1\theta_2}$$

7.3 Soundness

The soundness of the lifted calculus is a relatively straightforward property. We would like to say that if $P \mathbin{/} S \mid \theta$ then $P\theta \mathbin{/} S\theta$. However, the co-domain of θ may contain free variables, so the latter may not be well defined. We therefore have to admit an arbitrary grounding substitution σ' to be composed with θ. In the proof, we also need to extend σ' to account

for additional variables. We write $\sigma'' \supseteq \sigma'$ for an extension of σ' with some additional pairs t/x for ground terms t.

Theorem 7.1 *If $A \;/\; S \mid \theta$ then for any substitution σ' with $\mathrm{FV}((A/S)\theta\sigma') = \emptyset$ we have $A\theta\sigma' \;/\; S\theta\sigma'$.*

Proof: By induction on the structure of $\mathcal{D}$ of $P \;/\; S \mid \theta$.

Cases: The first three rule for conjunction and truth are straightforward and omitted here.

Case: $\mathcal{D} = \dfrac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \qquad \begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ \mathrm{cod}(\rho) \cap \mathrm{FV}(P/S) = \emptyset \\ P'\rho \doteq P \mid \theta_1\end{array} \qquad \begin{array}{c}\mathcal{D}' \\ B'\rho\theta_1 \;/\; S\theta_1 \mid \theta_2\end{array}}{P \;/\; S \mid \theta_1\theta_2}$

where $A = P$ and $\theta = \theta_1\theta_2$.

$\mathrm{FV}(P(\theta_1\theta_2)\sigma') = \mathrm{FV}(S(\theta_1\theta_2)\sigma') = \emptyset$	Assumption
Choose $\sigma'' \supseteq \sigma'$ such that $\mathrm{FV}((B'\rho\theta_1)\theta_2\sigma'') = \emptyset$	
$(B'\rho\theta_1)\theta_2\sigma'' \;/\; (S\theta_1)\theta_2\sigma''$	By i.h. on $\mathcal{D}'$
$B'(\rho\theta_1\theta_2\sigma'') \;/\; S\theta_1\theta_2\sigma''$	By assoc. of composition
$P'\rho\theta_1 = P\theta_1$	By soundness of unification
$P'\rho\theta_1\theta_2\sigma'' = P\theta_1\theta_2\sigma''$	By equality reasoning
$P'(\rho\theta_1\theta_2\sigma'') = P\theta_1\theta_2\sigma''$	By assoc. of composition
$P\theta_1\theta_2\sigma'' \;/\; S\theta_1\theta_2\sigma''$	By rule (using $\tau = \rho\theta_1\theta_2\sigma''$)
$P(\theta_1\theta_2)\sigma'' \;/\; S(\theta_1\theta_2)\sigma''$	By assoc. of composition
$P(\theta_1\theta_2)\sigma' \;/\; S(\theta_1\theta_2)\sigma'$	Since $\sigma' \subseteq \sigma''$ and $\mathrm{FV}(P(\theta_1\theta_2)\sigma') = \mathrm{FV}(S(\theta_1\theta_2)\sigma') = \emptyset$

□

The fact that we allow an arbitrary grounding substitution in the statement of the soundness theorem is not just technical device. It means that if there are free variables left in the answer substitution θ, then any instance of θ is also a valid answer. For example, if we ask `append([1,2,3], Ys, Zs)` and obtain the answer `Zs = [1,2,3|Ys]` then just by substituting `[4,5]` for `Ys` we can conclude `append([1,2,3], [4,5], [1,2,3,4,5])` without any further search.

Unfortunately, in the presence of free variables built-in extra-logical Prolog predicates such as disequality, negation-as-failure, or cut destroy this property (in addition to other problems with soundness).

7.4 Completeness

Completness follows the blueprint in the completeness proof for unification. In the literature this is often called the *lifting lemma*, showing that if there is ground deduction of a judgment, there must be a more general free variable deduction.

The first try at a lifting lemma, in analogy with a similar completness property for unification, might be:

If $A\sigma$ / $S\sigma$ then A / $S \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.

This does not quite work for a technical reason: during proof search additional variables are introduced which could appear in the domain of θ (and therefore in the domain of $\theta\sigma'$), while σ does not provide a substitution term for them.

We can overcome this inaccuracy by just postulating that additional term/variable pairs can be dropped, written as $\sigma \subseteq \theta\sigma'$. In the theorem and proof below we always assume that substitutions τ and σ, possibly subscripted or primed, are *ground substitutions*, that is, their co-domain is empty.

Theorem 7.2 *If $A\sigma$ / $S\sigma$ for ground $A\sigma$, $S\sigma$, and σ, then A / $S \mid \theta$ and $\sigma \subseteq \theta\sigma'$ for some θ and ground σ'.*

Proof: The proof is by induction on the structure of the given deduction of $A\sigma$ / $S\sigma$.

Cases: The cases for the three rule for conjunction and truth are straightforward and omitted here.

Case: $\mathcal{D} = \dfrac{\begin{array}{lll} & \mathrm{dom}(\tau) = \mathbf{x} & \\ & \mathrm{cod}(\tau) = \emptyset & \mathcal{D}' \\ \forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma & P'\tau = P\sigma & B'\tau \;/\; S\sigma \end{array}}{P\sigma \;/\; S\sigma}$ where $A\sigma = P\sigma$.

$\tau = \rho\tau'$ for some renaming ρ and substitution τ'	
with $\mathrm{dom}(\rho) = \mathbf{x}$, $\mathrm{cod}(\rho) = \mathrm{dom}(\tau')$, and	
$\mathrm{dom}(\tau') \cap \mathrm{dom}(\sigma) = \emptyset$	Choosing fresh vars.
(τ', σ) a valid substitution	By disjoint domains
$(P'\rho)\tau' = (P'\rho)(\tau', \sigma)$	$\mathrm{dom}(\sigma) \cap \mathrm{FV}(P'\rho) = \emptyset$
$S\sigma = S(\tau', \sigma)$	$\mathrm{dom}(\tau') \cap \mathrm{FV}(S) = \emptyset$
$P\sigma = P(\tau', \sigma)$	$\mathrm{dom}(\tau') \cap \mathrm{FV}(P) = \emptyset$

$P'\tau = P\sigma$	Given premiss
$P'\rho(\tau', \sigma) = P(\tau', \sigma)$	By equality reasoning
$P'\rho \doteq P \mid \theta_1$ and	
$(\tau', \sigma) = \theta_1\sigma'_1$ for some θ_1 and σ'_1	By completness of unification
$B'\tau \;/\; S\sigma$	Given subderivation $\mathcal{D}'$
$B'\tau = B'\rho\tau'$	By equality reasoning
$= B'\rho(\tau', \sigma)$	$\mathrm{dom}(\sigma) \cap \mathrm{FV}(B'\rho) = \emptyset$
$= B'\rho(\theta_1\sigma'_1)$	By equality reasoning
$= (B'\rho\theta_1)\sigma'_1$	By assoc. of composition
$S\sigma = S(\tau', \sigma) = S(\theta_1\sigma'_1)$	By equality reasoning
$= (S\theta_1)\sigma'_1$	By assoc. of composition
$B'\rho\theta_1 \;/\; S\theta_1 \mid \theta_2$ and	
$\sigma'_1 \subseteq \theta_2\sigma'_2$ for some θ_2 and σ'_2	By i.h. on $\mathcal{D}'$
$P \;/\; S \mid \theta_1\theta_2$	By rule
$\sigma \subseteq (\tau', \sigma) = \theta_1\sigma'_1$	By equality reasoning
$\subseteq \theta_1(\theta_2\sigma'_2)$	$\mathrm{cod}(\theta_1\sigma'_1) = \emptyset$
$= (\theta_1\theta_2)\sigma'_2$	By assoc. of composition

□

7.5 Occurs-Check Revisited

Now that the semantics of proof search with free variables has been clarified, we return to the issue that Prolog omits the occurs-check as mentioned in the last lecture. Instead, it builds circular terms when encountering problems such as $X \doteq f(X)$.

To understand why, we reconsider the `append` program.

```
append(nil, Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

In order to append two ground lists (for simplicity we assume of the same length), we would issue a query

```
?- append([x1, ..., xn], [y1, ..., yn], Zs).
```

The fact that the first clause does not apply is discovered in one or two steps, because nil clashes with cons. We then copy the second clause and unify `[X|Xs] = [`$x_1, \ldots, x_n$`]`. Assuming all the x_i are integers, this operation will still take $O(n)$ operations because of the occurs-check when unifying `Xs = [`$x_2, \ldots, x_n$`]`. Similarly, unifying `Y = [`$y_1, \ldots, y_n$`]` would

take $O(n)$ steps to perform the occurs-check. Finally the unification in the last argument `[X|Zs1] = Zs` just takes constant time.

Then the recursive call looks like

```
?- append([x2, ..., xn], [y1, ..., yn], Zs1).
```

which again takes $O(n)$ operations. Overall, we will recurse $O(n)$ times, performing $O(n)$ operations on each call, giving us a complexity of $O(n^2)$. Obviously, this is unacceptable for a simple operations such as appending two lists, which should be $O(n)$.

We can see that the complexity of this implementation is almost entirely due to the occurs-check. If we do not perform it, then a query such as the one in our example will be $O(n)$.

However, I feel the price of soundness is too high. Fortunately, in practice, the occurs-check can often be eliminated in a sound interpreter or compiler. The first reason is that in the presence of mode information, we may know that some argument in the goal are ground, that is, contain no variables. In that case, the occurs-check is entirely superfluous.

Another reason is slightly more subtle. As we can see from the operational semantics, we copy the clause (and the clause head) by applying a renaming substitution. The variables in the renamed clause head, $P'\rho$ are entirely new and are not allowed do not appear in the goal P or the goal stack S. As a result, we can omit the occurs-check when we first encounter a variable in the clause head, because that variable couldn't possible occur in the goal.

However, we have to be careful for the second occurrence of a variable. Consider the goal

```
?- append([], [1|Xs], Xs).
```

Clearly, this should fail because there is no term t such that $[1|t] = t$. If we unify with the clause head `append([], Ys, Ys)`, the first unification `Ys = [1|Xs]` can be done without the occurs-check.

However, after the substitution for `Ys` has been carried out, the third argument to append yields the problem `[1|Xs] = Xs` which can only be solved correctly if the occurs-check is carried out.

If your version of Prolog does not have switch to enable sound unification to be used in its operational semantics, you can achieve the same effect using the built-in `unify_with_occurs_check/2`. For example, we can rewrite `append` to the following sound, but ugly program.

```
append(nil, Ys, Zs) :- unify_with_occurs_check(Ys, Zs).
append([X|Xs], Ys, [Z|Zs]) :-
        unify_with_occurs_check(X, Z),
        append(Xs, Ys, Zs).
```

7.6 Historical Notes

The basic idea of lifting to go from a ground deduction to one with free variables goes back to Robinson's seminal work on unification and resolution [3], albeit in the context of theorem proving rather than logic programming. The most influential early paper on the theory of logic programming is by Van Emden and Kowalski [2], who introduced several model-theoretic notions that I have replaced here by more flexible proof-theoretic definitions and relate them to each other. An important completeness result regarding the subgoal selection strategy was presented by Apt and Van Emden [1] which can be seen as an analogue to the completeness result we presented.

7.7 Exercises

Exercise 7.1 *Give ground and free variable forms of deduction in a formulation without an explicit goal stack, but with an explicit program, and show soundness and completeness of the free variable version.*

Exercise 7.2 *Fill in the missing cases in the soundness proof for free variable deduction.*

Exercise 7.3 *Give three concrete counterexamples showing that the substitution property for free variables in an answer substitution fails in the presence of disequality on non-ground goals, negation-as-failure, and cut.*

Exercise 7.4 *Fill in the missing cases in the completeness proof for free variable deduction.*

Exercise 7.5 *Analyze the complexity of appending two ground lists of integers of length n and k given the optimization that the first occurrence of a variable in a clause head does not require an occurs-check. Then analyze the complexity if it is known that the first two argument to* append *are ground. Which one is more efficient?*

7.8 References

[1] Krzysztof R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, July 1982.

[2] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[3] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

15-819K: Logic Programming

Lecture 8

Completion

Frank Pfenning

September 21, 2006

In this lecture we extend the ground backtracking semantics to permit free variables. This requires a stronger normal form for programs. After introducing this normal form related to the so-call *iff-completion* of a program, we give the semantics at which point we have a complete specification of the pure Prolog search behavior including unification, subgoal selection and backtracking. At this point we return to the logical meaning of Prolog programs and derive the iff-completion of a program via a process called *residuation*.

8.1 Existential Quantification

First we return to the backtracking semantics with the intent of adding free variables and unification to make it a fully specified semantics for pure Prolog.

The first problem is presented by the rules for atomic goals. In a ground calculus, the rule

$$\frac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \qquad \begin{array}{l}\mathrm{dom}(\tau) = \mathbf{x} \\ \mathrm{cod}(\tau) = \emptyset \\ P'\tau = P\end{array} \qquad B'\tau \;/\; S \;/\; F}{P \;/\; S \;/\; F}$$

is correct only if we stipulate that for every ground atomic goal P there is *exactly one* program clause for which the rule above can be applied. Otherwise, not all failure or choice points would be explicit in the semantics.

Now we need the property that for every atomic goal P (potentially containing free variables) there is *exactly one* program clause that applies. Because goals can have the form $p(X_1, \ldots, X_n)$ for variables $X_1, \ldots, X_n$, this means that for every predicate p there should be only one clause in the program. Moreover, the head of this clause must unify with *every* permissible goal. At first this may seem far-fetched, but since our extended language includes equality, we can actually achieve this by transforming the program so that all clause heads have the form $p(X_1, \ldots, X_n)$ for distinct variables $X_1, \ldots, X_n$.

As an example, consider the `member` predicate in the form appropriate for the ground semantics.

```
member(X, []) :- fail.
member(X, [Y|Ys]) :- X = Y ; member(X, Ys).
```

We can transform this further by factoring the two cases as

```
member(X, Ys) :-
        (Ys = [], fail) ;
        (Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

This can be simplified, because the first disjunct will always fail.

```
member(X, Ys) :- Ys = [Y|Ys1], (X = Y ; member(X, Ys1)).
```

Writing such a program would be considered poor style, since the very first one is much easier to read. However, as an internal representation it turns out to be convenient.

We take one more step which, unfortunately, does not have a simple rendering in all implementations of Prolog. We can simplify the treatment of atomic goals further if the only free variables in a clause are the ones appearing in the head. In the `member` example above, this is not the case, because `Y` and `Ys1` occur in the body, but not the head. If we had existential quantification $\exists x.\, A$, we could overcome this. In logical form, the rule would be the following.

$$\frac{\exists y.\, \exists ys_1.\, Ys \doteq [y|ys_1] \wedge (X \doteq y \vee \mathsf{member}(X, ys_1))\ \mathit{true}}{\mathsf{member}(X, Ys)\ \mathit{true}}$$

In some implementations of Prolog, existential quantification is available with the syntax `X^A` for $\exists x.\, A$.[1] Then the program above would read

[1] You should beware, however, that some implementations of this are unsound in that the variable `X` is visible outside its scope.

```
member(X, Ys) :-
        Y^Ys1^(Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

On the logical side, This requires a new form of proposition, $\exists x.\, A$, where x is a *bound variable* with scope A. We assume that we can always rename bound variables. For example, we consider $\exists x.\, \exists y.\, p(x, x, y)$ and $\exists y.\, \exists z.\, p(y, y, z)$ to be identical. We also assume that bound variables (written as lower-case identifiers) and free variables (written as upper-case identifiers) are distinct syntactic classes, so no clash between them can arise.

Strictly speaking we should differentiate between substitutions for variables x that may be bound, and for logic variables X (also called meta-variables). At this point, the necessary machinery for this distinction would yield little benefit, so we use the same notations and postulate the same properties for both.

Existential quantification is now defined by

$$\frac{A(t/x)\ true}{\exists x.\, A\ true}\ \exists I$$

where the the substitution term t is understood to have no free variables since logical deduction is ground deduction.

When carrying out a substitution θ, we must take care when encountering a quantifier $\exists x.\, A$. If x is in the domain of θ, we should first rename it to avoid possible confusion between the bound x and the x that θ substitutes for. The second condition is that x does not occur in the co-domain of θ. This condition is actually vacuous here (t is closed), but required in more general settings.

$$(\exists x.\, A)\theta \quad = \quad \exists x.\, (A\theta) \quad \text{provided } x \notin \mathrm{dom}(\theta) \cup \mathrm{cod}(\theta)$$

Recall the convention that bound variables can always be silently renamed, so we can always satisfy the side condition no matter what θ is.

The search semantics for existential quantification comes in two flavors: in the ground version we guess the correct term t, in the free variable version we substitute a fresh logic variable.

$$\frac{A(t/x)\ /\ S}{\exists x.\, A\ /\ S} \qquad \frac{A(X/x)\ /\ S \mid \theta \quad X \notin \mathrm{FV}(\exists x.\, A\ /\ S)}{\exists x.\, A\ /\ S \mid \theta}$$

Since X does not occur in $\exists x.\, A\ /\ S$, we could safely restrict θ in the conclusion to remove any substitution term for X.

8.2 Backtracking Semantics with Free Variables

Now we assume the program is in a normal form where for each atomic predicate p of arity n there is exactly one clause

$$\forall x_1 \ldots \forall x_n.\, p(x_1, \ldots x_n) \leftarrow B'$$

where the $\mathrm{FV}(B') \subseteq \{x_1, \ldots, x_n\}$.

We will not endeavor to return the answer substitution from the free variable judgment, but just describe the computation to either success or failure. The extension to compute an answer substitution requires some thought, but does not add any essentially new elements (see Exercise 8.1). Therefore, the we write $A \;/\; S \;/\; F$ where A, S, and F may have free variables. The intended interpretation is that if $A \;/\; S \;/\; F$ in the free variable semantics then there exists a grounding substitution σ such that $A\sigma \;/\; S\sigma \;/\; F\sigma$ in the ground semantics. This is not very precise, but sufficient for our purposes.

First, the rules for conjunction and truth. They are the same in the free and ground semantics.

$$\frac{A \;/\; B \wedge S \;/\; F}{A \wedge B \;/\; S \;/\; F} \qquad \frac{B \;/\; S \;/\; F}{\top \;/\; B \wedge S \;/\; F} \qquad \frac{}{\top \;/\; \top \;/\; F}$$

Second, the rules for disjunction and falsehood. Again, it plays no role if the goals are interpreted as closed or with free variables.

$$\frac{A \;/\; S \;/\; (B \wedge S) \vee F}{A \vee B \;/\; S \;/\; F} \qquad \frac{B \;/\; S' \;/\; F}{\bot \;/\; S \;/\; (B \wedge S') \vee F} \qquad \begin{array}{c}\textit{fails (no rule)}\\ \bot \;/\; S \;/\; \bot\end{array}$$

Third, the equality rules. Clearly, these involve unification, so substitutions come into play.

$$\frac{t \doteq s \mid \theta \quad \top \;/\; S\theta \;/\; F}{t \doteq s \;/\; S \;/\; F} \qquad \frac{\text{there is no } \theta \text{ with } t \doteq s \mid \theta \quad \bot \;/\; S \;/\; F}{t \doteq s \;/\; S \;/\; F}$$

When unification succeeds, the most general unifier θ must be applied to the success continuation S which shares variables with t and s. But we do not apply the substitution to F. Consider a goal of the form $(X \doteq a \wedge p(X)) \vee (X \doteq b \wedge q(X))$ to see that while we try the first disjunction, $X \doteq a$ the instantiations of X should not affect the failure continuation.

The rule for existentials just introduces a globally fresh logic variable.

$$\frac{A(X/x) \;/\; S \;/\; F \quad X \notin \mathrm{FV}(\exists x.\, A \;/\; S \;/\; F)}{\exists x.\, A \;/\; S \;/\; F}$$

Finally the rule for atomic propositions. Because of the normal form for each clause in the program, this rule no longer involves unification or generation of fresh variables. Such considerations have now been relegated to the cases for equality and existential quantification.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B') \in \mathcal{P} \quad B'(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F}{p(\mathbf{t}) \;/\; S \;/\; F}$$

Here we wrote $\mathbf{t}/\mathbf{x}$ as an abbreviation for the substitution $t_1/x_1, \ldots, t_n/x_n$ where $\mathbf{t} = t_1, \ldots, t_n$ and $\mathbf{x} = x_1, \ldots, x_n$.

8.3 Connectives as Search Instructions

The new operational semantics, based on the normal form for programs, beautifully isolates various aspects of the operational reading for logic programs. It is therefore very useful as an intermediate form for compilation.

Procedure Call ($p(\mathbf{t})$). An atomic goal $p(\mathbf{t})$ now just becomes a procedure call, interpreting a predicate p as a procedure in logic programming. We use a substitution $\mathbf{t}/\mathbf{x}$ for parameter passing if the clause is $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B'$ and $\mathrm{FV}(B') \subseteq \mathbf{x}$.

Success ($\top$). A goal $\top$ simply succeeds, signaling the current subgoal has been solved. Stacked up subgoals are the considered next.

Conjunctive choice ($A \wedge B$). A conjunction $A \wedge B$ represents two subgoals that have to be solved. The choice is which one to address first. The rule for conjunction says A.

Failure ($\bot$). A goal $\bot$ simply fails, signaling the current subgoal fails. We backtrack to previous choice points, exploring alternatives.

Disjunctive choice ($A \vee B$). A disjunction $A \vee B$ represents a choice between two possibly path towards a solution. The rules for disjunction say we try A first and later B (if A fails).

Existential choice ($\exists x.\, A$). An existential quantification $\exists x.\, A$ represents the choice which term to use for x. The rule for existential quantification says to postpone this choice and simply instantiate x with a fresh logic variable to be determined later during search by unification.

Unification ($t \doteq s$). An equality $t \doteq s$ represents a call to unification, to determine some existential choices postponed earlier in a least committed way (if a unifier exists) or fail (if no unifier exists).

Let us read the earlier `member` program as a sequence of these instructions.

```
member(X, Ys) :-
        Y^Ys1^(Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

Given a *procedure call*

```
?- member(t, s).
```

we substitute actual arguments for formal parameters, reaching

```
?- Y^Ys1^(s = [Y|Ys1], (t = Y ; member(t, Ys1))).
```

We now *create fresh variables* `Y` and `Ys1` (keeping their names for simplicity), yielding

```
?- (s = [Y|Ys1], (t = Y ; member(t, Ys1))).
```

Now we have *two subgoals* to solve (a conjunction), which means we solve the left side first by *unifying* s, the second argument in the call to `member`, with `[Y|Ys1]`. If this fails, we fail and backtrack. If this succeeds, we apply the substitution to the second subgoal.

Let us assume $s = [s_1|s_2]$, so that the substitution will be $s_1/Y, s_2/Y s_1$. Then we have to solve

```
?- t = s1 ; member(t, s2).
```

Now we have a *disjunctive choice*, so we first try to *unify* t with s_1, pushing the alternative onto the failure continuation. If unification succeeds, we succeed with the unifying substitution. Note that besides the unifying substitution, we have also changed the failure continuation by pushing a call to `member` onto it. If the unification fails, we try instead the second alternative, calling `member` recursively.

```
?- member(t, s2).
```

I hope this provides some idea how the body of the `member` predicate could be compiled to a sequence of instructions for an abstract machine.

8.4 Logical Semantics Revisited

So far, we have carefully defined truth for all the connectives and quantifers except for universal quantification and implication which appeared only in the program. These introduction rules show how to establish that a given proposition is true. For universal quantification and implication we follow a slight different path, because from the logical point of view the program is a set of *assumptions*, not something we are trying to prove. In the language of judgments, we are dealing with a so-called *hypothetical judgment*

$$\Gamma \vdash A \ \mathit{true}$$

where Γ represents the program. It consists of a collection of propositions $D_1 \ \mathit{true}, \ldots, D_n \ \mathit{true}$.

In logic programming, occurrences of logical connectives are quite restricted. In fact, as we noted at the beginning, pure Prolog has essentially no connectives, just atomic predicates and inference rules. We have only extended the language in order to accurately describe search behavior in a logical notation. The restrictions are different for what is allowed as a program clause and what is allowed as a goal. So for the remainder of this lecture we will use G to stand for legal goals and D to stand for legal program propositions.

We summarize the previous rules in this slightly generalized form.

$$\frac{\Gamma \vdash G_1 \ \mathit{true} \quad \Gamma \vdash G_2 \ \mathit{true}}{\Gamma \vdash G_1 \wedge G_2 \ \mathit{true}} \wedge I \qquad \frac{}{\Gamma \vdash \top \ \mathit{true}} \top I$$

$$\frac{\Gamma \vdash G_1 \ \mathit{true}}{\Gamma \vdash G_1 \vee G_2 \ \mathit{true}} \vee I_1 \qquad \frac{\Gamma \vdash G_2 \ \mathit{true}}{\Gamma \vdash G_1 \vee G_2 \ \mathit{true}} \vee I_2 \qquad \frac{\textit{no rule}}{\Gamma \vdash \bot \ \mathit{true}}$$

$$\frac{}{\Gamma \vdash t \doteq t \ \mathit{true}} \doteq I \qquad \frac{\Gamma \vdash G(t/x) \ \mathit{true}}{\Gamma \vdash \exists x.\, G \ \mathit{true}} \exists I$$

When the goal is atomic, we have to use an assumption, corresponding to a clause in the program. Choosing a particular assumption and then breaking down its structure as an assumption is called *focusing*. This is a new judgment $\Gamma; D \ \mathit{true} \vdash P \ \mathit{true}$. The use of the semi-colon here ";" is unrelated to its use in Prolog where it denotes disjunction. Here is just isolates a particular assumption D. We call this the *focus* rule.

$$\frac{D \in \Gamma \quad \Gamma; D \ \mathit{true} \vdash P \ \mathit{true}}{\Gamma \vdash P \ \mathit{true}} \text{ focus}$$

When considering which rules should define the connectives in D it is important to keep in mind that the rules now define the *use* of an assumption, rather than how to prove its truth. Such rules are called *left rules* because the apply to a proposition to the left of the turnstile symbol '$\vdash$'.

First, if the assumption is an atomic fact, it must match the conclusion. In that case the proof is finished. We call this the *init* rule for *initial sequent*.

$$\frac{}{\Gamma; P\ true \vdash P\ true}\ \text{init}$$

Second, if the assumption is an implication we would have written $P \leftarrow B$ so far. We observe that B will be a subgoal, so we write it as G. Further, P does not need to be restricted to be an atom—it can be an arbitrary legal program formula D. Finally, we turn around the implication into the more customary form $G \supset D$.

$$\frac{\Gamma; D\ true \vdash P\ true \quad \Gamma \vdash G\ true}{\Gamma; G \supset D\ true \vdash P\ true}\ \supset L$$

Here, G actually appears as a *subgoal* in one premise and D as an assumption in the other, which is a correct given the intuitive meaning of implication to represent clauses.

If we have a universally quantified proposition, we can instantiate it with an arbitrary (closed) term.

$$\frac{\Gamma; D(t/x)\ true \vdash P\ true}{\Gamma; \forall x.\, D\ true \vdash P\ true}\ \forall L$$

It is convenient to also allow conjunction to combine multiple clauses for a given predicate. Then the use of a conjunction reduces to a choice about which conjunct to use, yielding two rules.

$$\frac{\Gamma; D_1\ true \vdash P\ true}{\Gamma; D_1 \wedge D_2\ true \vdash P\ true}\ \wedge L_1 \qquad \frac{\Gamma; D_2\ true \vdash P\ true}{\Gamma; D_1 \wedge D_2\ true \vdash P\ true}\ \wedge L_2$$

We can also allow truth, but there is no rule for it as an assumption

$$\frac{\textit{no rule}}{\Gamma; \top\ true \vdash P\ true}$$

since the assumption that $\top$ is true gives us no information for proving P.

We have explicitly *not* defined how to *use* disjunction, falsehood, equality, or existential quantification, or how to *prove* implication or universal quantification. This is because attempting to add such rules would significantly change the nature of logic programming. From the rules, we can read off the restriction to goals and program as conforming to the following grammar.

$$\begin{array}{lrcl} \text{Goals} & G & ::= & P \mid G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid t \doteq s \mid \exists x.\, G \\ \text{Clauses} & D & ::= & P \mid D_1 \wedge D_2 \mid \top \mid G \supset D \mid \forall x.\, D \end{array}$$

Clauses in this form are equivalent to so-called *Horn clauses*, which is why it is said that Prolog is based on the Horn fragment of first-order logic.

8.5 Residuation

A program in the form described above is rather general, but we can transform it into the procedure call form described earlier with a straightforward and elegant algorithm. To begin, for any predicate p we collect all clauses contributing to the definition of p into a single proposition D_p, which is the conjunction of the universal closure of the clauses whose head has the form $p(\mathbf{t})$. For example, given the program

```
nat(z).
nat(s(N)) :- nat(N).

plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

we generate a logical rendering in the form of two propositions:

$$\begin{array}{rcl} D_{\mathsf{nat}} & = & \mathsf{nat}(\mathsf{z}) \wedge \forall n.\, \mathsf{nat}(n) \supset \mathsf{nat}(\mathsf{s}(n)), \\ D_{\mathsf{plus}} & = & (\forall n.\, \mathsf{plus}(\mathsf{z}, n, n)) \\ & & \quad \wedge \forall m.\, \forall n.\, \forall p.\, \mathsf{plus}(m, n, p) \supset \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \end{array}$$

The idea now is that instead of playing through the choices for breaking down D_p when searching for a proof of

$$\Gamma; D_p \vdash p(\mathbf{t})$$

we residuate those choices into a goal whose search behavior is equivalent. If we write the residuating judgment as

$$D_p \vdash p(\mathbf{x}) > G$$

then the focus rule would be

$$\frac{D_p \in \Gamma \quad D_p \vdash p(\mathbf{x}) > G_p \quad \Gamma \vdash G_p(\mathbf{t}/\mathbf{x})}{\Gamma \vdash p(\mathbf{t})}$$

Residuation must be done *deterministically* and is not allowed to fail, so that we can view G_p as the compilation of $p(\mathbf{x})$, the parametric form of a call to p.

To guide the design of the rules, we will want that if $D_p \vdash p(\mathbf{x}) > G$ then $\Gamma; D_p \vdash p(\mathbf{t})$ iff $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$. Further more, if D_p and $p(\mathbf{x})$ are given, then there exists a unique G such that $D_p \vdash p(\mathbf{x}) > G$.

$$\frac{}{p'(\mathbf{s}) \vdash p(\mathbf{x}) > p'(\mathbf{s}) \doteq p(\mathbf{x})} \qquad \frac{D_1 \vdash p(\mathbf{x}) > G_1 \quad D_2 \vdash p(\mathbf{x}) > G_2}{D_1 \wedge D_2 \vdash p(\mathbf{x}) > G_1 \vee G_2}$$

$$\frac{}{\top \vdash p(\mathbf{x}) > \bot} \qquad \frac{D \vdash p(\mathbf{x}) > G_1}{G \supset D \vdash p(\mathbf{x}) > G_1 \wedge G}$$

$$\frac{D \vdash p(\mathbf{x}) > G \quad y \notin \mathbf{x}}{\forall y.\, D \vdash p(\mathbf{x}) > \exists y.\, G}$$

The side condition in the last rule can always be satisfied by renaming of the bound variable x.

First, the soundness of residuation. During the proof we will discover a necessary property of deductions, called a *substitution property*. You may skip this and come back to it once you understands its use in the proof below.

Lemma 8.1 *If $D \vdash p(\mathbf{x}) > G$ and $y \notin \mathbf{x}$, then $D(s/y) \vdash p(\mathbf{x}) > G(s/y)$ for any closed term s. Moreover, if the original derivation is $\mathcal{D}$, the resulting derivation $\mathcal{D}(s/y)$ has exactly the same structure as $\mathcal{D}$.*

Proof: By induction on the structure of the derivation for $D \vdash p(\mathbf{x}) > G$. In each case we just apply the induction hypothesis to all premisses and rebuild the same deduction from the results. □

Now we can prove the soundness.

Theorem 8.2 *If $D \vdash p(\mathbf{x}) > G$ for $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$ and $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$ for ground $\mathbf{t}$ then $\Gamma; D \vdash p(\mathbf{t})$.*

Proof: By induction on the structure of the deduction of the given residuation judgment, applying inversion to the second given deduction in each case. All cases are straightforward, except for the case of quantification which we show.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ D_1 \vdash p(\mathbf{x}) > G_1 \quad y \notin \mathbf{x}\end{array}}{\forall y.\, D_1 \vdash p(\mathbf{x}) > \exists y.\, G_1}$ where $D = \forall y.\, D_1$ and $G = \exists y.\, G_1$.

$\Gamma \vdash (\exists y.\, G_1)(\mathbf{t}/\mathbf{x})$	Assumption
$\Gamma \vdash \exists y.\, G_1(\mathbf{t}/\mathbf{x})$	Since $y \notin \mathbf{x}$ and $\mathbf{t}$ ground
$\Gamma \vdash G_1(\mathbf{t}/\mathbf{x})(s/y)$ for some ground s	By inversion
$\Gamma \vdash G_1(s/y)(\mathbf{t}/\mathbf{x})$	Since $y \notin \mathbf{x}$ and $\mathbf{t}$ and s ground
$D_1(s/y) \vdash p(\mathbf{x}) > G_1(s/y)$	By substitution property for residuation
$\Gamma; D_1(s/y) \vdash p(\mathbf{t})$	By i.h. on $\mathcal{D}_1(s/y)$
$\Gamma; \forall y.\, D_1 \vdash p(\mathbf{t})$	By rule

We may apply the induction hypothesis to $\mathcal{D}_1(s/y)$ because $\mathcal{D}_1$ is a subdeduction of $\mathcal{D}$, and $\mathcal{D}_1(s/y)$ has the same structure as $\mathcal{D}_1$.

□

Completeness follows a similar pattern.

Theorem 8.3 *If $D \vdash p(\mathbf{x}) > G$ with $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$ and $\Gamma; D \vdash p(\mathbf{t})$ for ground $\mathbf{t}$ then $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$*

Proof: By induction on the structure of the given residuation judgment, applying inversion to the second given deduction in each case. In the case for quantification we need to apply the substitution property for residuation, similarly to the case for soundness. □

Finally, termination and uniqueness.

Theorem 8.4 *If D and $p(\mathbf{x})$ are given with $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$, then there exists a unique G such that $D \vdash p(\mathbf{x}) > G$.*

Proof: By induction on the structure of D. There is exactly one rule for each form of D, and the propositions are smaller in the premisses. □

8.6 Logical Optimization

Residuation provides an easy way to transform the program into the form needed for the backtracking semantics with free variables. For each predicate p we calculate

$$D_p \vdash p(\mathbf{x}) > G_p$$

and then replace D_p by

$$\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p.$$

With respect to the focusing semantics, D_p is equivalent to the new formulation (see Exercise 8.4).

We reconsider the earlier example.

$$\begin{array}{lcl} D_{\mathsf{nat}} & = & \mathsf{nat}(\mathsf{z}) \wedge \forall n.\, \mathsf{nat}(n) \supset \mathsf{nat}(\mathsf{s}(n)), \\ D_{\mathsf{plus}} & = & (\forall n.\, \mathsf{plus}(\mathsf{z}, n, n)) \\ & & \quad \wedge \forall m. \forall n. \forall p.\, \mathsf{plus}(m, n, p) \supset \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \end{array}$$

Running our transformation judgment, we find

$$D_{\mathsf{nat}} \vdash \mathsf{nat}(x) > \mathsf{nat}(\mathsf{z}) \doteq \mathsf{nat}(x) \vee \exists n.\, \mathsf{nat}(\mathsf{s}(n)) \doteq \mathsf{nat}(x) \wedge \mathsf{nat}(n)$$

$$\begin{array}{l} D_{\mathsf{plus}} \vdash \mathsf{plus}(x_1, x_2, x_3) > (\exists n.\, \mathsf{plus}(\mathsf{z}, n, n) \doteq \mathsf{plus}(x_1, x_2, x_3)) \vee \\ \quad (\exists m. \exists n. \exists p.\, \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \doteq \mathsf{plus}(x_1, x_2, x_3) \wedge \mathsf{plus}(m, n, p)). \end{array}$$

These compiled forms can now be the basis for further simplification and optimizations. For example,

$$G_{\mathsf{plus}} = (\exists n.\, \mathsf{plus}(\mathsf{z}, n, n) \doteq \mathsf{plus}(x_1, x_2, x_3)) \vee \ldots$$

Given our knowledge of unification, we can simplify this equation to three equations.

$$G'_{\mathsf{plus}} = (\exists n.\, \mathsf{z} = x_1 \wedge n = x_2 \wedge n = x_3) \vee \ldots$$

Since n does not appear in the first conjunct, we can push in the existential quantifier, postponing the creation of an existential variable.

$$G''_{\mathsf{plus}} = (\mathsf{z} = x_1 \wedge \exists n.\, n = x_2 \wedge n = x_3) \vee \ldots$$

The next transformation is a bit trickier, but we can see that there exists an n which is equal to x_2 and x_3 iff x_2 and x_3 are equal. Since n is a bound variable occurring nowhere else, we can exploit this observation to elimination n altogether.

$$G'''_{\mathsf{plus}} = (\mathsf{z} = x_1 \wedge x_2 = x_3) \vee \ldots$$

The optimized code will unify the first argument with z and, if this succeeds, unify the second and third arguments.

8.7 Iff Completion

We now revisit the normal form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p$. Since this represents the only way of succeeding in a proof of $p(\mathbf{t})$, we can actually turn the implication around, replacing it by an *if and only if* ($\leftrightarrow$)

$$\forall \mathbf{x}.\, p(\mathbf{x}) \leftrightarrow G_p.$$

Of course, this proposition is outside the fragment that is amenable to logic programming search (considering the right-to-left implication), but it has found some application in the notion of definitional reflection, where it is usually written as

$$\forall \mathbf{x}.\, p(\mathbf{x}) \stackrel{\triangle}{=} G_p.$$

This allows us to draw conclusions that the program alone does not permit, specifically about the falsehood of propositions.

We do not have the formal reasoning rules available to us at this point, but given the (slightly optimized) iff-completion of nat,

$$\forall x.\, \mathsf{nat}(x) \leftrightarrow \mathsf{z} \doteq x \vee \exists n.\, \mathsf{s}(n) \doteq x \wedge \mathsf{nat}(n)$$

we would be able to prove, explicitly with formal rules, that $\mathsf{nat}(\mathsf{a})$ is false for a new constant a, because a is neither equal to z nor to $\mathsf{s}(n)$ for some n.

Unfortunately the expressive power of the completion is still quite limited in that it is much weaker than induction.

8.8 Historical Notes

The notion of iff-completion goes back to Clark [2] who investigated notions of negation and their justification in the early years of logic programming.

The use of goal-directed search and focusing to explain logic programming goes back to Miller et al. [3], who tested various extensions of the logic presented here for suitability as the foundation for a logic programming language.

The use of residuation for compilation and optimization has been proposed by Cervesato [1], who also shows that the ideas are quite robust by addressing a much richer logic.

The notion of definitional reflection goes back to Schroeder-Heister [5] who also examined its relationship to completion [4]. More recently, reflection has been employed in a theorem prover derived from logic programming [6] in the Bedwyr system.

8.9 Exercises

Exercise 8.1 *Extend the free variable semantics with backtracking to explicitly return an answer substitution θ. State the soundness and completness of this semantics with respect to the one that does not explicitly calculate the answer substitution. If you feel brave, prove these two theorems.*

Exercise 8.2 *Prove that the free variable backtracking semantics given in lecture is sound and complete with respect to the ground semantics.*

Exercise 8.3 *Prove the completeness of residuation.*

Exercise 8.4 *Given soundness and completeness of residuation, show that if we replace programs D_p by $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p$ where $D_p \vdash p(\mathbf{x}) > G_p$ then the focusing semantics is preserved.*

8.10 References

[1] Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 115–129, Manchester, England, June 1998. MIT Press.

[2] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[3] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[4] Peter Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNCS 798, March 1993.

[5] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the 8th Annual Symposium on Logic in computer Science (LICS'93)*, pages 222–232, Montreal, Canada, July 1993. IEEE Computer Society Press.

[6] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In C.Benzmüller, J.Harrison, and C.Schürmann, editors, *Proceedings of the Workshop on Empirically Successful Automated Reasnoing in Higher-Order Logics (ESHOL'05)*, pages 79–98, Montego Bay, Jamaica, December 2005.

15-819K: Logic Programming

Lecture 9

Types

Frank Pfenning

September 26, 2006

In this lecture we introduce types into logic programming, primarily to distinguish meaningful from meaningless terms and propositions, thereby capturing errors early during program development.

9.1 Views on Types

Types are a multi-faceted concept, and also subject of much debate. We will discuss some of the views that have been advanced regarding the role of types in logic programming, and then focus on the one we find most useful and interesting.

Descriptive Types. In logic programming terminology, a type system is called *descriptive* if it captures some aspect of program behavior, where program behavior is defined entirely without reference to types. A common example is to think of a type as an approximation to the success set of the program, that is, the set of terms on which it holds. Reconsider addition on unary numbers:

```
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

We can see that if `plus(`t_1`,` t_2`,` t_3`)` is true according to this definition, then t_1 must be a natural number, that is, `nat(`t_1`)` according to the definition

```
nat(z).
nat(s(N)) :- nat(N).
```

On the other hand, nothing interesting can be said about the other two arguments, because the first clause as written permits non-sensical propositions such as `plus(z, [], [])` to be true.

The nature of descriptive types means that usually they are used to optimize program behavior rather than as an aid to the programmer for writing correct code.

Prescriptive Types. A type system is called *prescriptive* if it is an integral part of the meaning of programs. In the example above, if we prescribe that `plus` is a relation between three terms of type `nat`, then we suddenly differentiate well-typed expressions (such as `plus(z, s(z), P)` when `P` is a variable of type `nat`) from expressions that are not well-typed and therefore meaningless (such as `plus(z, [], [])`. Among the well-typed expressions we then further distinguish those propositions which are true and those which are false.

Prescriptive types, in a well-designed type system, are immediately useful to the programmer since many accidental mistakes in the program will be captured rather then leading to either failure or success, which may be very difficult to debug. In some ways, the situation in pure Prolog is worse than in other dynamically typed languages such as Lisp, because in the latter language you will often get a run-time error for incorrect programs, while in pure Prolog everything is either true or false. This is somewhat overstates the case, since many built-in predicates have dynamically enforced type restrictions. Nevertheless programming with no static and few dynamic errors becomes more and more difficult as programs grow larger. I recognize that it may be somewhat difficult to fully appreciate the point if you write only small programs as you do in this class (at least so far).

Within the prescriptive type approach, we can make further distinction as to the way types are checked or expressed.

Monadic Propositions as Types. In this approach types are represented as monadic predicates, such as `nat` above. This approach is prevalent in the early logic programming literature, and it is also the prevalent view classical logic. The standard answer to the question why the traditional predicate calculus (also known as classical first-order logic) is untyped is that types can be eliminated in favor of monadic predicates. For example, we can translate $\forall x{:}\mathsf{nat}.\, A$ into $\forall x.\, \mathsf{nat}(x) \supset A$, where x now ranges over arbitrary terms. Similarly, $\exists x{:}\mathsf{nat}.\, A$ becomes $\exists x.\, \mathsf{nat}(x) \wedge A$.

In a later lecture we will see that this view is somewhat difficult to sustain in the presence of higher-order predicates, that is, predicates that take other predicates as arguments. The corresponding *higher-order logic*, also known as Church classical theory of types, therefore has a different concept of type called *simple*. Fortunately, the two ideas are compatible, and it is possible to refine the simple type system using the ideas behind monadic propositions, but we have to postpone this idea to a future lecture.

Simple Types. In this approach types are explicitly declared as new entities, separately from monadic propositions. Constructors for types in the form of constants and function symbols are also separately declared. Often, types are disjoint so that a given term has a unique type. In the example above, we might declare

```
nat : type.
z : nat.
s : nat -> nat.
plus : nat, nat, nat -> o.
```

where `o` is a distinguished type of propositions. The first line declares `nat` to be a new type, the second and third declare `z` and `s` as constructors for terms of type `nat`, and the last line declares `plus` as a predicate relating three natural numbers.

With these declarations, an expression such as `plus(z, [], [])` can be readily seen as ill-typed, since the second and third argument are presumably of type `list` and not `nat`. Moreover, in a clause `plus(z, N, N)` it is clear that `N` is a variable ranging only over terms of type `nat`.

In this lecture we develop a system of simple types. One of the difficulty we encounter is that generic data structures, including even simple lists, are difficult to deal with unless we have a type of all terms, or permit variables types. For example, while lists of natural numbers are easy

```
natlist : type.
[] : natlist.
[_|_] : nat, natlist -> natlist.
```

there is no easy way to declare lists with elements of unknown or arbitrary type. We will address this shortcoming in the next lecture.

9.2 Signatures

Simple types rely on explicit declarations of new types, and of new constructors together with their type. The collection of such declarations is called a *signature*. We assume a special type constant o (omicron) that stands for the type of propositions. We use the letters τ and σ for newly declared types. These types will always be atomic and not include o.[1]

Signature	Σ	$::=$	$\cdot$	empty signature
		$\mid$	$\Sigma, \tau : \mathit{type}$	type declaration
		$\mid$	$\Sigma, f : \tau_1, \ldots, \tau_n \to \tau$	function symbol declaration
		$\mid$	$\Sigma, p : \tau_1, \ldots, \tau_n \to o$	predicate symbol declaration

As usual, we will abbreviate sequences of types by writing them in boldface, $\boldsymbol{\tau}$ or $\boldsymbol{\sigma}$. Also, when a sequence of types is empty we may abbreviate $c : \cdot \to \tau$ by simply writing $c : \tau$ and similarly for predicates. All types, functions, and predicates declared in a signature must be distinct so that lookup of a symbol is always unique.

Despite the suggestive notation, you should keep in mind that "$\to$" is not a first class constructor of types, so that, for example, $\tau \to \tau$ is *not* a type for now. The only true types we have are atomic type symbols. This is similar to the way we developed logic programming: the only propositions we had were atomic, and the logical connectives only came in later to describe the search behavior of logic programs.

9.3 Typing Propositions and Terms

There are three basic judgments for typing: one for propositions, one for terms, and one for sequences of terms. All three require a context Δ in which the types of the free variables in a proposition or term are recorded.

$$\text{Typing Context} \quad \Delta \quad ::= \quad \cdot \mid \Delta, x{:}\tau$$

We assume all variables in a context are distinct so that the type assigned to a variable is unique. We write $\mathrm{dom}(\Delta)$ for the set of variables declared in a context.

A context Δ represents *assumptions* on the types of variables and is therefore written on the left side of a turnstile symbol '$\vdash$', as we did with

[1] Allowing τ to be o would make the logic higher order, which we would like to avoid for now.

logic programs before.

$$\begin{array}{ll} \Sigma; \Delta \vdash A : o & A \text{ is a valid proposition} \\ \Sigma; \Delta \vdash t : \tau & \text{term } t \text{ has type } \tau \\ \Sigma; \Delta \vdash \mathbf{t} : \boldsymbol{\tau} & \text{sequence } \mathbf{t} \text{ has type sequence } \boldsymbol{\tau} \end{array}$$

Because the signature Σ never changes while type-checking, we omit it from the judgments below and just assume that there is a fixed signature Σ in the background theory.

The rules for propositions are straightforward. As is often the case, these rules, read bottom-up, have an interpretation as an algorithm for type-checking.

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \wedge B : o} \qquad \frac{}{\Delta \vdash \top : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \vee B : o} \qquad \frac{}{\Delta \vdash \bot : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \supset B : o} \qquad \frac{p : \boldsymbol{\tau} \to o \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash p(\mathbf{t}) : o}$$

For equality, we demand that the terms we compare have the same type τ, whatever that may be. So rather than saying that, for example, zero is not equal to the empty list, we consider such a question meaningless.

$$\frac{\Delta \vdash t : \tau \quad \Delta \vdash s : \tau}{\Delta \vdash t \doteq s : o}$$

In the rules for quantifiers, we have to recall the convention that bound variables can be renamed silently. This is necessary to ensure that the variable declarations we add to the context do not conflict with existing declarations.

$$\frac{\Delta, x{:}\tau \vdash A : o \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \forall x{:}\tau.\, A : o} \qquad \frac{\Delta, x{:}\tau \vdash A : o \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \exists x{:}\tau.\, A : o}$$

While we have given the condition $x \notin \mathrm{dom}(\Delta)$ in these two rules, in practice they are often omitted by convention.

Next we come to typing terms and sequences of terms.

$$\frac{f : \boldsymbol{\tau} \to \sigma \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash f(\mathbf{t}) : \sigma} \qquad \frac{x{:}\tau \in \Delta}{\Delta \vdash x : \tau}$$

$$\frac{\Delta \vdash t : \tau \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash (t, \mathbf{t}) : (\tau, \boldsymbol{\tau})} \qquad \frac{}{\Delta \vdash (\cdot) : (\cdot)}$$

This system does not model the overloading for predicate and function symbols at different arities that is permitted in Prolog. In this simple first-order language this is relatively easy to support, but left as Exercise 9.2.

9.4 Typing Substitutions

In order to integrate types fully into our logic programming language, we need to type all the artifacts of the operational semantics. Fortunately, the success and failure continuations are propositions for which typing is already defined, and the same is true for programs. This leaves substitutions, as calculated by unification.

For a substitution, we just demand that if we substitute t for x and x has type τ, the t also must have type τ. We write the judgment as $\Delta \vdash \theta \ \mathit{subst}$, expressing that θ is a well-typed substitution.

$$\frac{x{:}\tau \in \Delta \quad \Delta \vdash t : \tau}{\Delta \vdash (\theta, t/x) \ \mathit{subst}} \qquad \frac{}{\Delta \vdash (\cdot) \ \mathit{subst}}$$

Our prior conventions that all the variables defined by a substitution are distinct, and that the domain and codomain of a substitution are disjoint also still apply. Because we would like to separate typing from other considerations, they are not folded into the rules above which could easily be done.

9.5 Types and Truth

Now that we have extended our language to include types, we need to consider how this affects the various judgment we have. The most basic one is truth. Since this is defined for ground expression (that is, expression without free variables), we do not need to generalize this to carry a context Δ, although it will have to carry a signature Σ. We leave this implicit as in the presentation of the typing judgments.

We presuppose that any proposition we write is a well-typed proposition, that is, has type o. In other words, if we write $A\ true$ we implicitly assume that $\cdot \vdash A : o$. We have to be careful that our rules maintain this property, and in which direction the rule is read. For example, when the rule

$$\frac{\Gamma \vdash A\ true}{\Gamma \vdash A \vee B\ true}\ \vee I$$

is read from the premiss to the conclusion, then we would need a second premiss to check that $B : o$.

However, we prefer to read it as "*Assuming $A \vee B$ is a valid proposition, $A \vee B$ is true if A is true.*" In this reading, the condition on B is implicit.

Since the bottom-up reading of rules is pervasive in logic programming, we will adopt the same here. Then, only the rules for quantifiers require an additional typing premiss. Recall that Γ represents a fixed program.

$$\frac{\cdot \vdash t : \tau \quad \Gamma \vdash A(t/x)\ true}{\Gamma \vdash \exists x{:}\tau.\ A\ true}\ \exists I \qquad \frac{\cdot \vdash t : \tau \quad \Gamma; A(t/x)\ true \vdash P\ true}{\Gamma; \forall x{:}\tau.\ A \vdash P\ true}\ \forall L$$

This assumes that if we substitute a term of type τ for a variable of type τ, the result will remain well-typed. Fortunately, this property holds and is easy to prove.

Theorem 9.1 *Assume $\Delta \vdash \theta\ subst$.*

1. *If $\Delta \vdash t : \tau$ then $\Delta \vdash t\theta : \tau$.*
2. *If $\Delta \vdash \mathbf{t} : \boldsymbol{\tau}$ then $\Delta \vdash \mathbf{t}\theta : \boldsymbol{\tau}$.*
3. *If $\Delta \vdash A : o$ then $\Delta \vdash A\theta : o$.*

Proof: By mututal induction on the structure of the given typing derivations for t, $\mathbf{t}$, and A. □

9.6 Type Preservation

A critical property tying together a type system with the operational semantics for a programming language is *type preservation*. It expresses that if we start in a well-typed state, during the execution of a program all intermediate states will be well-typed. This is an absolutely fundamental property without which a type system does not make much sense. Either the type system or the operational semantics needs to be revised in such a case so that they match at least to this extent.

$$\frac{\Delta \vdash G_1 \;/\; G_2 \wedge S \;/\; F}{\Delta \vdash G_1 \wedge G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \wedge S \;/\; F}{\Delta \vdash \top \;/\; G_2 \wedge S \;/\; F} \qquad \frac{}{\Delta \vdash \top \;/\; \top \;/\; F}$$

$$\frac{\Delta \vdash G_1 \;/\; S \;/\; (G_2 \wedge S) \vee F}{\Delta \vdash G_1 \vee G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \;/\; S' \;/\; F}{\Delta \vdash \bot \;/\; S \;/\; (G_2 \wedge S') \vee F} \qquad \begin{array}{c}\textit{fails (no rule)}\\ \Delta \vdash \bot \;/\; S \;/\; \bot\end{array}$$

$$\frac{\Delta \vdash t \doteq s \mid \theta \quad \Delta \vdash \top \;/\; S\theta \;/\; F}{\Delta \vdash t \doteq s \;/\; S \;/\; F} \qquad \frac{\begin{array}{c}\text{there is no } \theta \text{ with}\\ \Delta \vdash t \doteq s \mid \theta \end{array} \quad \Delta \vdash \bot \;/\; S \;/\; F}{\Delta \vdash t \doteq s \;/\; S \;/\; F}$$

$$\frac{\Delta, x{:}\tau \vdash G \;/\; S \;/\; F \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \exists x{:}\tau.\, G \;/\; S \;/\; F}$$

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G) \in \Gamma \quad \Delta \vdash G(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F}{\Delta \vdash p(\mathbf{t}) \;/\; S \;/\; F}$$

Figure 1: Operational Semantics Judgment

It is worth stepping back to make explicit in which way the inference rules for our last judgment (with goal stack, failure continuation, and free variables) constitute a transition system for an abstract machine. We will add a context Δ to the judgment we had so far in order to account for the types of the free variables. For, in the form of inference rules from last lecture. We assume the clauses for each predicate p are in a normal form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B'$, all collected in a fixed program Γ, and that a signature Σ is also fixed.

Each rule with one premise can be seen as a state transition rule. For example, the very first rule becomes

$$(\Delta \vdash A \wedge B \;/\; S \;/\; F) \Rightarrow (\Delta \vdash A \;/\; B \wedge S \;/\; F).$$

We do not write down the others, which can be obtained by simple two-dimensional rearrangement.

The state $\Delta \vdash \top \;/\; \top \;/\; F$ is a final state (success) since the corresponding rule has no premiss, as is the state $\Delta \vdash \bot \;/\; S \;/\; \bot$ (failure) since there is no corresponding rule.

The rules for equality have two premisses and make up two conditional

transition rules.

$$\begin{array}{lcll} (\Delta \vdash t \doteq s \;/\; S \;/\; F) & \Rightarrow & (\Delta \vdash \top \;/\; S\theta \;/\; F) & \text{provided } \Delta \vdash t \doteq s \mid \theta \\ (\Delta \vdash t \doteq s \;/\; S \;/\; F) & \Rightarrow & (\Delta \vdash \bot \;/\; S\theta \;/\; F) & \text{provided there is no } \theta \text{ with} \\ & & & \Delta \vdash t \doteq s \mid \theta \end{array}$$

In order to state type preservation, we need a judgment of typing for a state of the abstract machine, $\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}$. It is defined by a single rule.

$$\frac{\Delta \vdash G : o \quad \Delta \vdash S : o \quad \Delta \vdash F : o}{\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}}$$

In addition, we assume that G, S, and F have the shape of a goal, goal stack, and failure continuation, respectively, following this grammar:

$$\begin{array}{lcll} \text{Goals} & G & ::= & G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid t \doteq s \mid \exists x{:}\tau.\, G \mid p(\mathbf{t}) \\ \text{Goal Stacks} & S & ::= & \top \mid G \wedge S \\ \text{Failure Conts} & F & ::= & \bot \mid (G \wedge S) \vee F \end{array}$$

The preservation theorem now shows that if state s is valid and $s \Rightarrow s'$, then s' is valid. To prove this we first need a lemma about unification.

Theorem 9.2 *If $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ and $\Delta \vdash s \doteq t \mid \theta$ then $\Delta \vdash \theta \; \mathit{subst}$. Similarly, if $\Delta \vdash \mathbf{t} : \boldsymbol{\tau}$ and $\Delta \vdash \mathbf{s} : \boldsymbol{\tau}$ and $\Delta \vdash \mathbf{s} \doteq \mathbf{t} \mid \theta$ then $\Delta \vdash \theta \; \mathit{subst}$.*

Proof: By mutual induction on the structures of $\mathcal{D}$ of $\Delta \vdash s \doteq t \mid \theta$ and $\mathcal{D}'$ of $\Delta \vdash \mathbf{s} \doteq \mathbf{t} \mid \theta$, applying inversion to the derivations of $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ as needed. □

Now we can state (and prove) preservation.

Theorem 9.3 *If $\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}$ and $(\Delta \vdash G \;/\; S \;/\; F) \Rightarrow (\Delta' \vdash G' \;/\; S' \;/\; F')$ then $\Delta' \vdash G' \;/\; S' \;/\; F' \; \mathit{state}$.*

Proof: First we apply inversion to conclude that G, S, and F are all well-typed propositions. Then we distinguish cases on the transition rules, applying inversion to the typing derivations for G, S, and F as needed to reassemble the derivations that G', S', and F' are also well-typed.

In the case for unification we appeal to the preceding lemma, and the lemma that applying well-typed substitutions preserves typing.

In the case for existential quantification, we need an easy lemma that we can always add a new typing assumption to a given typing derivation. □

9.7 The Phase Distinction

While the operational semantics (including unification) preserves types, it does not refer to them during execution. In that sense, the types appearing with quantifiers or the context Δ are not necessary to execute programs. This means that our type system obeys a so-called *phase distinction*: we can type-check our programs, but then execute programs without further reference to types. This is a desirable property since it tells us that there is no computational overhead to types at all. On the contrary: types could help a compiler to optimize the program because it does not have to account for the possibility of ill-typed expressions. Types also help us to sort out ill-formed expressions, but they do not change the meaning or operational behavior of the well-formed ones. Generally, as type systems become more expressive, this property is harder to maintain as we will see in the next lecture.

9.8 Historical Notes

Type systems have a long and rich history, having been developed originally to rule out problems such as Russell's paradox [3] in the formulation of expressive logics for the formalization of mathematics. Church's theory of types [1] provided a great simplification and is also known as classical higher-order logic. It uses a type o (omicron) of propositions, a single type ι (iota) for individuals, and closes types under function spaces.

In logic programming, the use of types was slow to arrive since the predicate calculus (its logical origin) does not usually employ them. Various articles on notion of types in logic programming are available in an edited collection [2].

9.9 Exercises

Exercise 9.1 *The notion of type throws a small wrinkle on the soundness and non-deterministic completeness of an operational semantics with free variables. The new issue is the presence of possibly empty types, either because there are no constructors, or the constructors are such that no ground terms can be formed. Discuss the issues.*

Exercise 9.2 *Write out an extension of the system of simple types given here which permits Prolog-like overloading of function and predicate symbols at different arity. Your extension should continue to satisfy the preservation theorem.*

Exercise 9.3 *Write a Prolog program for type checking propositions and terms. Extend your program to type inference where the types on quantifiers are not explicit, generating an explicitly typed proposition (if it is indeed well-typed).*

Exercise 9.4 *Rewrite the operational rules so that unification is explicitly part of the transitions for the abstract machine, rather than a condition.*

Exercise 9.5 *Show the cases for unification, existential quantification, and atomic goals in the proof of the type preservation theorem in detail.*

Exercise 9.6 *Besides type preservation, another important property for a language is* progress*: any well-formed state of an abstract machine is either an explicitly characterized final state, or can make a further transition.*

In our language, this holds with or without types, if we declare states $\top$ / $\top$ / F (for any F) and $\bot$ / S / $\bot$ (for any S) final, and assume that G, S, and F satisfy the grammar for goals, goal stacks and failure continuations shown in this lecture.

Prove the progress theorem.

9.10 References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[3] Bertrand Russell. Letter to Frege. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 124–125. Harvard University Press, 1967. Letter written in 1902.

15-819K: Logic Programming

Lecture 10

Polymorphism

Frank Pfenning

September 28, 2006

In this lecture we extend the system of simple types from the previous lecture to encompass polymorphism. There are some technical pitfalls, and some plausible systems do not satisfy type preservation. We discuss three ways to restore type preservation.

10.1 Heterogeneous and Homogeneous Lists

Types such as natural numbers, be it in binary or in unary notation, are easy to specify and use in the system from the previous lecture. Generic data structures such as lists, on the other hand, present difficulties. Recall the type predicate for lists:

```
list([]).
list([X|Xs]) :- list(Xs).
```

The difficulty is that for lists in general there is no restriction on `X`: it can have arbitrary type. When we try to give the declarations

```
list : type.
[] : list.
'.' : ?, list -> list.
```

we realize that there is nothing sensible we can put as the type of the first argument of cons.[1]

Two solutions suggest themselves. One is to introduce a universal type "any" and ensure that $t : any$ for all terms t. This destroys the property of

[1] Recall that `[X|Xs]` is just alternate syntax for `'.'(X, Xs)`

simple types that every well-typed term has a unique type and significantly complicates the type system. Following this direction it seems almost inevitable that some types need to be carried at runtime. A second possibility is to introduce type variables and think of the type of constructors as schematic in their free type variables.

```
list : type.
[] : list.
'.' : A, list -> list.
```

We will pursue this idea in this lecture. Although it is also not without problems, it is quite general and leads to a rich and expressive types system.

Since the typing rules above are schematic, we get to choose a fresh instance for `A` every time we use the cons constructor. This means the elements of a list can have arbitrarily different types (they are *heterogeneous*).

For certain programs it is important to know that the elements of a list all have the same type. For example, we can sort a list of integers, but not a list mixing integers, booleans, and other lists. This requires that `list` is actually a type constructor: it takes a type as an argument and returns a type. Specifically:

```
list : type -> type.
[] : list(A).
'.' : A, list(A) -> list(A).
```

With these declarations only homogeneous lists will type-check: a term of type `list(A)` will be a list all of whose elements are of type `A`.

10.2 Polymorphic Signatures

We now move to a formal specification of typing. We start with signatures, which now have a more general form. We write α for type variables and $\boldsymbol{\alpha}$ for a sequences of type variables. As in the case of clauses and ordinary variables, the official syntax quantifies over type variables in declarations.

Signature	Σ	$::=$	$\cdot$	empty signature
		$\mid$	$\Sigma, a : \mathbf{type}_n \to \mathit{type}$	type constructor declaration
		$\mid$	$\Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$	function symbol declaration
		$\mid$	$\Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o$	predicate symbol declaration

Here, boldface "$\mathbf{type}_n$" stands for a sequence $\mathit{type}, \ldots, \mathit{type}$ of length n. As usual, if a sequence to the left of the arrow is empty, we may omit

the arrow altogether. Similarly, we may omit the quantifier if there are no type variables, and the argument to a type zero-ary types constructor $a()$. Moreover, function and predicate declarations should not contain any free type variables.

The language of types is also more elaborate, but still does not contain function types as first-class constructors.

$$\text{Types} \quad \tau \quad ::= \quad \alpha \mid a(\tau_1, \ldots, \tau_n)$$

10.3 Polymorphic Typing for Terms

The only two rules in the system for simple types that are affected by polymorphism are those for function and predicate symbols. We account for the schematic nature of function and predicate declarations by allowing a substitution $\hat{\theta}$ for the type variables $\boldsymbol{\alpha}$ that occur in the declaration. We suppose a fixed signature Σ.

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

We use the notation $\hat{\theta}$ to indicate a substitution of types for type variables rather terms for term variables.

Looking ahead (or back) at the required property of type preservation, one critical lemma is that unification produces a well-typed substitution. Unfortunately, in the presence of polymorphic typing, this property fails! You may want to spend a couple of minutes thinking about a possible counterexample before reading on. One way to try to find one (and also a good start on fixing the problem) is to attempt a proof and learn from its failure.

False Claim 10.1 *If $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ and $\Delta \vdash t \doteq s \mid \theta$ then $\Delta \vdash \theta$ subst and similarly for sequences of terms.*

Proof attempt: We proceed by induction on the derivation $\mathcal{D}$ of the unification judgment, applying inversion to the given typing judgments in each case. We focus on the problematic one.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \Delta \vdash \mathbf{t} \doteq \mathbf{s} \mid \theta\end{array}}{\Delta \vdash f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta}$.

We note that we could complete this case if we could appeal to the induction hypothesis on $\mathcal{D}'$, since this would yield the well-typedness of θ. We can appeal to the induction hypothesis if we can show that $\mathbf{t}$ and $\mathbf{s}$ have the same sequence of types. Let's see what we can glean from applying inversion to the giving typing derivations. First, we note that there must be a *unique* type declaration for f in the signature, say

$$f : \boldsymbol{\sigma} \to \tau' \in \Sigma \text{ for some } \boldsymbol{\sigma} \text{ and } \tau'.$$

Now we write out the inversions on the given typing derivations, using the uniqueness of the declaration for f.

$\Delta \vdash f(\mathbf{t}) : \tau$	Assumption
$\tau = \tau'\hat{\theta}_1$ and $\Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}_1$ for some $\hat{\theta}_1$	By inversion
$\Delta \vdash f(\mathbf{s}) : \tau$	Assumption
$\tau = \tau'\hat{\theta}_2$ and $\Delta \vdash \mathbf{s} : \boldsymbol{\sigma}\hat{\theta}_2$ for some $\hat{\theta}_2$	By inversion

At this point we would like to conclude

$$\boldsymbol{\sigma}\hat{\theta}_1 = \boldsymbol{\sigma}\hat{\theta}_2$$

because then $\mathbf{t}$ and $\mathbf{s}$ would have the same sequence of types and we could finish this case by the induction hypothesis.

Unfortunately, this is not necessarily the case because all we know is

$$\tau = \tau'\hat{\theta}_1 = \tau'\hat{\theta}_2.$$

From this we can only onclude that θ_1 and θ_2 agree on the type variables free in τ', but they could differ on variables that occur only in $\boldsymbol{\sigma}$ but not in τ'.

$\diamond$

From this we can construct a counterexample. Consider heterogeneous lists

$$\begin{array}{lcl} \mathsf{nil} & : & \mathsf{list} \\ \mathsf{cons} & : & \forall\alpha.\, \alpha, \mathsf{list} \to \mathsf{list} \end{array}$$

Then

$$\begin{array}{lcl} x{:}\mathsf{nat} \vdash \mathsf{cons}(x, \mathsf{nil}) & : & \mathsf{list} \\ x{:}\mathsf{nat} \vdash \mathsf{cons}(\mathsf{nil}, \mathsf{nil}) & : & \mathsf{list} \end{array}$$

and

$$x{:}\mathsf{nat} \vdash \mathsf{cons}(x, \mathsf{nil}) \doteq \mathsf{cons}(\mathsf{nil}, \mathsf{nil}) \mid (\mathsf{nil}/x)$$

but the returned substitution (nil/x) is not well typed because x:nat and nil:list.

Because unification does not return well-typed substitutions, the operational semantics in whichever form we presented does also not preserve types. The design of the type system is flawed.

We can pursue two avenues to fix this problem: restricting the type system or rewriting the operational semantics.

Type Restriction. In analyzing the failed proof above we can see that at least this case would go through if we require for a declaration $f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$ that every type variable that occurs in $\boldsymbol{\sigma}$ also occurs in τ. Function symbols of this form are called *type preserving*.[2] When all function symbols are type preserving, the falsely claimed property above actually does hold—the critical case is the one we gave.

Requiring all function symbols to be type preserving rules out heterogeneous lists, and we need to apply techniques familiar from functional programming to inject elements into a common type. I find this tolerable, but many Prolog programmers would disagree.

Type Passing. We can also modify the operational semantics so that types are passed and unified at run-time. Then we can use the types to prevent the kind of failure of preservation that arose in the counterexample above. Passing types violates the phase separation and therefore has some overhead. On the other hand, it allows data structures such as heterogeneous lists without additional coding. The language λProlog uses a type passing approach, together with some optimizations to avoid unnecessary passing of types. We return to this option below.

Before we can make a choice between the two, or resolve the apparent conflict, we must consider the type preservation theorem to make sure we understand all the issues.

[2] Function symbols are constructors, so this is not the same as type preservation in a functional language. Because of this slightly unfortunate terminology this property has also been called *transparent*.

10.4 Polymorphic Predicates

As it turns out, requiring all function symbols to be type preserving is insufficient to guarantee type preservation. The problem is presented by predicate symbols, declared now as $p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o$. If we just unify $\Delta \vdash p(\mathbf{t}) \doteq p(\mathbf{s})$, we run into the same problem as above because predicate symbols are *never* type preserving unless they do not have any type variables at all.

Disallowing polymorphic predicates altogether would be too restrictive, because programs that manipulate generic data structures must be polymorphic. For example, for homogeneous lists we have

```
append : list(A), list(A), list(A) -> o.
```

which is polymorphic in the type variable `A`.

Moreover, the program clauses themselves also need to be polymorphic. For example, in the first clause for `append`

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

the variable `Ys` should be of type `list(A)` for a type variable `A`.

Before we can solve our problem with type preservation, we must account for the presence of type variables in program clauses, which now quantify not only over term variables, but also over type variables. The general form is $\forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{t}) \leftarrow G$ where the type variables $\boldsymbol{\alpha}$ contain all free type variables in the clause. We will come back to the normal form used in the free variables operational semantics below.

The meaning of universal quantification over types is specified via substitution in the focusing judgment.

$$\frac{\Gamma; D(\tau/\alpha)\ \mathit{true} \vdash P\ \mathit{true}}{\Gamma; \forall \alpha.\, D\ \mathit{true} \vdash P\ \mathit{true}}$$

This just states that an assumption that is schematic in a type variable α can be instantiated to any type τ. Since logical deduction is ground, we implicitly assume in the rule above that τ does not contain any free type variables.

This forces a new typing rule for clauses, which in turn means we have to slightly generalize contexts to permit declarations $\alpha\ \mathit{type}$ for type variables.

$$\frac{\Delta, \alpha\ \mathit{type} \vdash A : o}{\Delta \vdash \forall \alpha.\, A : o}$$

To be complete and precise, we also need a judgment that types themselves are well-formed ($\Delta \vdash \tau \ \mathit{type}$) and similarly for type substitutions ($\Delta \vdash \hat{\theta} \ \mathit{tsubst}$); they can be found in various figures at the end of these notes.

$$\frac{\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \qquad p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma \quad \Delta \vdash \hat{\theta} \ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}}{\Delta \vdash p(\mathbf{t}) : o}$$

Now we reconsider how to solve the problem of polymorphic predicates. Looking at the predicate `append`

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

for homogenous lists, we see that no run-time type error can actually arise. This is because the heads of the clauses cover the most general case for a goal. For example, `Ys` has type `list(A)` for an arbitrary type `A`. It can therefore take on the type of the list in the goal. For example, with a goal

```
append([], [1,2,3], Zs).
```

with `Zs : list(int)`, no problem arises because we instantiate the clause to `A = int` and then use that instance.

On the other hand, if we added a clause

```
append([1], [], [1]).
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

which is certainly not logically incorrect albeit redundant, then a run-time type error would occur if we called it with a goal such as

```
?- append([X], [], Zs), plus(X, s(z), s(s(z))).
```

where `X : nat`.

What is required is that each clause head for a predicate p is *maximally general* or *parametric* with respect to the declaration of p. More precisely, we say a clause

$$\forall \boldsymbol{\beta}.\, \forall \mathbf{x}{:}\boldsymbol{\tau}.\, p(\mathbf{t}) \leftarrow G \qquad \text{with} \qquad p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma$$

is *parametric* if there is a type substitution $\hat{\theta}$ with $\mathrm{dom}(\hat{\theta}) = \boldsymbol{\beta}$ with $\boldsymbol{\alpha}\ \mathbf{type} \vdash \hat{\theta} \ \mathit{tsubst}$ such that

$$\boldsymbol{\alpha}\ \mathbf{type}, \mathbf{x}{:}\boldsymbol{\tau}\hat{\theta} \vdash \mathbf{t} : \boldsymbol{\sigma}.$$

In other words, the types of the arguments $\mathbf{t}$ to p in the clause head must match $\boldsymbol{\sigma}$ in their full generality, keeping all types $\boldsymbol{\alpha}$ fixed.

The proof of type preservation in the presence of this restriction is technically somewhat involved[3] so we will omit it here. In the next section we will see an alternative approach for which type preservation is easy.

10.5 Polymorphic Residuation

When previously describing residuation, we actually were somewhat cavalier in the treatment of atomic programs (that is, clause heads).

$$\frac{}{p'(\mathbf{s}) \vdash p(\mathbf{x}) > p'(\mathbf{s}) \doteq p(\mathbf{x})}$$

Strictly speaking, the resulting equation is not well-formed because it relates two atomic propositions rather than two terms. We can eliminate this inaccuracy by introducing equality between term sequences as a new goal proposition, writing is as $\mathbf{t} \doteq \mathbf{s}$ in overloaded notation. Then we can split the residuation above into two rules:

$$\frac{}{p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

The new pair of rules removes equality between propositions. However, the first rule now has the problem that if p is not maximally general, the residuated equation $\mathbf{s} \doteq \mathbf{x}$ may not be well-typed!

The idea now is to do some checking during residuation, so that it fails when a clause head is not maximally general. Since residuation would normally be done statically, as part of compilation of a logic program, we discover programs that violate the condition at compile time, before the program is executed. Following this idea requires passing in a context Δ. to residuation so we can perform type-checking. The two rules above then become

$$\frac{\Delta \vdash \mathbf{x} : \boldsymbol{\sigma} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\sigma}}{\Delta; p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{\Delta; p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

For the program fragment D_p defining p with

$$p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o$$

[3] As I am writing these notes, I do not have a complete and detailed proof in the present context.

we initially invoke residuation with the most general atomic goal $p(\mathbf{x})$ as in

$$\boldsymbol{\alpha}\ \mathbf{type}, \mathbf{x}{:}\boldsymbol{\sigma}; D_p \vdash p(\mathbf{x}) > G_p$$

leading to the residuated program clause

$$\forall\boldsymbol{\alpha}.\, \forall\mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p.$$

Residuation is easily extended to handle universally quantified type variables in programs: we just have to guess how to instantiate them so that when reaching the head the arguments have the types $\boldsymbol{\sigma}$.

$$\frac{\Delta \vdash \tau\ \mathit{type} \quad \Delta; D(\tau/\beta) \vdash p(\mathbf{x}) > G}{\Delta; \forall\beta.\, D \vdash p(\mathbf{x}) > G}$$

Note that this residuation never generates any residual existential quantification over types. This means that the operational semantics should not allow any free type variables at run-time. This makes sense from a practical perspective: even though programs are generic in the types of data they can manipulate, when we execute programs they operate on concrete data. Moreoever, since we do not actually carry the types around, their role would be quite unclear. Nevertheless, an extension to residuate types that cannot be guessed at compile-time is possible (see Exercise 10.1).

The operational semantics refers to the residuated program. Since it contains no equations involving predicates, and we assume all function symbols are type preserving, type preservation is now a relatively straightforward property. The rule for predicate invocation looks as follows:

$$\frac{\begin{array}{l}(\ p : \forall\boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o\ ) \\ (\ \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}(\boldsymbol{\tau}/\boldsymbol{\alpha})\ ) \\ \forall\boldsymbol{\alpha}.\, \forall\mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p \qquad \Delta \vdash G_p(\boldsymbol{\tau}/\boldsymbol{\alpha})(\mathbf{t}/\mathbf{x})\ /\ S\ /\ F\end{array}}{\Delta \vdash p(\mathbf{t})\ /\ S\ /\ F}$$

In this rule, formally, we determine a type substitution $\boldsymbol{\tau}/\boldsymbol{\alpha}$, but this is only a technical device in order to make it easier to state and prove the type preservation theorem. We have indicated this by parenthesizing the extraneous premisses. During the actual operation of the abstract machine, quantifiers are not annotated with types, and type substitutions are neither computed nor applied.

Now type preservation follows in a pretty straightforward way. A critical lemma is *parametricity over types*[4]: if $\boldsymbol{\alpha}\ \mathbf{type}, \Delta \vdash t : \sigma$ then $\Delta(\boldsymbol{\tau}/\boldsymbol{\alpha}) \vdash$

[4]This notion is related to, but not the same as the semantic notion of parametricity in functional programming.

$t : \sigma(\boldsymbol{\tau}/\boldsymbol{\alpha})$ and similarly for proposition. Essentially, if we keep a type variable fixed in a typing derivation, we can substitute an arbitrary type for the variable and still get a proper derivation. This can be proven easily by induction over the structure of the given typing derivation.

We also have that if $\Delta \vdash D : o$ and $\Delta \vdash p(\mathbf{x}) : o$ for a program proposition D and $\Delta; D \vdash p(\mathbf{x}) > G$ then $\Delta \vdash G : o$. Of course, this theorem is possible precisely because we check types in the case where D is atomic.

For reference, we recap some of the judgments and languages. We have simplified equalities by using only equality of term sequences, taking the case of single terms as a special case. Recall that $G \supset D$ and $D \leftarrow G$ are synonyms.

Signatures	Σ	::=	$\cdot \mid \Sigma, a : \mathbf{type}_n \rightarrow \mathit{type} \mid \Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow \tau$
			$\mid \Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o$
Contexts	Δ	::=	$\cdot \mid \Delta, \alpha\ \mathit{type} \mid \Delta, x{:}\tau$
Types	τ	::=	$a(\boldsymbol{\tau}) \mid \alpha$
Programs	Γ	::=	$\cdot \mid \Gamma, \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p$
Clauses	D	::=	$p(\mathbf{t}) \mid D_1 \wedge D_1 \mid \top \mid G \supset D \mid \forall x{:}\tau.\, D \mid \forall \alpha.\, D$
Goals	G	::=	$p(\mathbf{t}) \mid G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid \mathbf{t} \doteq \mathbf{s} \mid \exists x{:}\tau.\, G$
Goal Stacks	S	::=	$\top \mid G \wedge S$
Failure Conts	F	::=	$\bot \mid (G \wedge S) \vee F$

Rules defining typing and well-formedness judgments on these expressions are given at the end of these notes. Programs are in normal form so that they can be used easily in a backtracking free-variable semantics.

Now we can state and prove the type preservation theorem in its polymorphic form.

Theorem 10.2 *Assume a well-formed signature Σ, program Γ, and context Δ. Further assume that all function symbols are type preserving. If $\Delta \vdash G \ / \ S \ / \ F$ state and $(\Delta \vdash G \ / \ S \ / \ F) \Rightarrow (\Delta' \vdash G' \ / \ S' \ / \ F')$ then $\Delta' \vdash G' \ / \ S' \ / \ F'$ state.*

Proof: By distinguishing cases on the transition relation, applying inversion on the given derivations. In some cases, previously stated lemmas such as the soundness of unification and preservation of types under well-formed substitutions are required. □

10.6 Parametric and Ad Hoc Polymorphism

The restrictions on function symbols (type preserving) and predicate definitions (parametricity) imply that no types are necessary during the execu-

tion of well-typed programs.

The condition that clause heads must be maximally general implies that the programs behave parametrically in their type. The `append` predicate, for example, behaves identically for lists of all types. This is also a characteristic of parametric polymorphism in functional programming, so we find the two conditions neatly relate the two paradigms.

On the other hand, the parametricity restriction can be somewhat unpleasant on occasion. For example, we may want a generic predicate `print` that dispatches to different, more specialized predicates, based on the type of the argument. This kind of predicate is *not* parametric and in its full generality would require a type passing interpretation. This is a form of *ad hoc polymorphism* which is central in object-oriented languages.

We only very briefly sketch how *existential types* might be introduced to permit a combination of parametric polymorphism with ad hoc polymorphism, the latter implemented with type passing.

For each function symbol, we shift the universal quantifiers that do not appear in the result type into an existential quantifier over the arguments. That is,

$$f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$$

is transformed into

$$f : \forall \boldsymbol{\alpha}_1.\, (\exists \boldsymbol{\alpha}_2.\, \boldsymbol{\sigma}) \to \tau$$

where $\boldsymbol{\alpha} = \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2$ and $\boldsymbol{\alpha}_1 = FV(\tau)$. By the latter condition, the declaration can now be considered type preserving.

To make this work with the operational semantics we apply f not just to terms, but also to the types corresponding to $\boldsymbol{\alpha}_2$. For example, heterogeneous lists

```
list : type.
nil : list.
cons : A, list -> list.
```

are interpreted as

$$\begin{array}{lcl} \mathsf{list} & & \textit{type} \\ \mathsf{nil} & : & \mathsf{list} \\ \mathsf{cons} & : & (\exists \alpha.\, \alpha, \mathsf{list}) \to \mathsf{list} \end{array}$$

A source level term

```
cons(1, cons(z, nil))
```

would be represented as

$$\mathsf{cons}(\mathsf{int}; 1, \mathsf{cons}(\mathsf{nat}; \mathsf{z}, \mathsf{nil}))$$

where we use a semi-colon to separate type arguments from term arguments. During unification, the type argument as well as the term arguments must be unified to guarantee soundness.

For predicates, type parameters that are not treated parametrically must be represented as existential quantifiers over the arguments, the parametric ones remain universal. For example,

```
append([1], [], [1]).
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

for homogeneous lists is ad hoc polymorphic because of the first clause and should be given type

$$\mathsf{append} \quad : \quad (\exists\alpha.\, \mathsf{list}(\alpha), \mathsf{list}(\alpha), \mathsf{list}(\alpha)) \to o$$

At least internally, but perhaps even externally, `append` now has one type argument and three term arguments.

```
append(int; [1], [], [1]).
append(A; [], Ys, Ys).
append(A; [X|Xs], Ys, [X|Zs]) :- append(A; Xs, Ys, Zs).
```

Unification of the type arguments (either implicitly or explicitly) prevents violation of type preservation, as can be seen from the earlier counterexample

```
?- append(nat; [X], [], Zs), plus(X, s(z), s(s(z))).
```

which no fails to match the first clause of `append`.

The extension of polymorphic typing by using type-passing existential quantifiers is designed to have the nice property that if the program is parametric and function symbols are type-preserving, then no types are passed at runtime. However, if function symbols or predicates are needed which violate these restrictions, they can be added with some feasible and local overhead.[5]

[5] At the point of this writing this is speculation—I have not seen, formally investigated, or implemented such a system.

10.7 Historical Notes

The first proposal for parametrically polymorphic typing in Prolog was made by Mycroft and O'Keefe [3] and was strongly influenced by the functional language ML. Hanus later refined and extended this proposal [1]. Among other things, Hanus considers a type passing interpretation for ad hoc polymorphic programs and typed unification. The modern dialect λProlog [2] incorporates polymorphic types and its Teyjus implementation contains several sophisticated optimizations to handle run-time types efficiently [4].

10.8 Exercises

Exercise 10.1 *Define an extension of residuation in the presence of polymorphism that allows free type variable in the body of a clause.*

Exercise 10.2 *Write out the rules for typing, unification, operational semantics, and sketch type preservation for a type-passing interpretation of existential types as outlined in this lecture.*

Exercise 10.3 *Write a type-checker for polymorphic Prolog programs, following the starter code and instructions available on the course website.*

10.9 References

[1] Michael Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[2] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[3] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, July 1984.

[4] Gopalan Nadathur and Xiaochu Qi. Optimizing the runtime processing of types in a higher-order logic programming language. In G. Suffcliff and A. Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, pages 110–125, Montego Bay, Jamaica, December 2005. Springer LNAI 3835.

10.10 Appendix: Judgment Definitions

We collect the rules for various judgments here for reference. General assumptions apply without being explicitly restated, such as the uniqueness of declarations in signatures, contexts, and substitutions, or tacit renaming of bound variables (both at the type and the term level). Also, many judgment implicitly carry a signature or program which never changes, so we elide them from the rules. If necessary, they are shown explicitly on the left of the judgment, separated by a semi-colon from other hypotheses.

$$\frac{\Delta \vdash \mathbf{x} : \boldsymbol{\sigma} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\sigma}}{\Delta; p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{\Delta; p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

$$\frac{\Delta; D_1 \vdash p(\mathbf{x}) > G_1 \quad \Delta; D_2 \vdash p(\mathbf{x}) > G_2}{\Delta; D_1 \wedge D_2 \vdash p(\mathbf{x}) > G_1 \vee G_2}$$

$$\frac{}{\Delta; \top \vdash p(\mathbf{x}) > \bot} \qquad \frac{\Delta; D \vdash p(\mathbf{x}) > G_1}{\Delta; G \supset D \vdash p(\mathbf{x}) > G_1 \wedge G}$$

$$\frac{\Delta, y{:}\tau; D \vdash p(\mathbf{x}) > G}{\Delta; \forall y{:}\tau.\, D \vdash p(\mathbf{x}) > \exists y{:}\tau.\, G}$$

$$\frac{\Delta \vdash \tau \ \mathit{type} \quad \Delta; D(\tau/\beta) \vdash p(\mathbf{x}) > G}{\Delta; \forall \beta.\, D \vdash p(\mathbf{x}) > G}$$

Figure 1: Residuation Judgment $\Sigma; \Delta; D \vdash p(\mathbf{x}) > G$

$$\frac{\Delta \vdash G_1 \;/\; G_2 \wedge S \;/\; F}{\Delta \vdash G_1 \wedge G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \wedge S \;/\; F}{\Delta \vdash \top \;/\; G_2 \wedge S \;/\; F} \qquad \frac{}{\Delta \vdash \top \;/\; \top \;/\; F}$$

$$\frac{\Delta \vdash G_1 \;/\; S \;/\; (G_2 \wedge S) \vee F}{\Delta \vdash G_1 \vee G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \;/\; S' \;/\; F}{\Delta \vdash \bot \;/\; S \;/\; (G_2 \wedge S') \vee F} \qquad \begin{array}{c}\textit{fails (no rule)}\\ \Delta \vdash \bot \;/\; S \;/\; \bot\end{array}$$

$$\frac{\Delta \vdash \mathbf{t} \doteq \mathbf{s} \mid \theta \quad \Delta \vdash \top \;/\; S\theta \;/\; F}{\Delta \vdash \mathbf{t} \doteq \mathbf{s} \;/\; S \;/\; F} \qquad \frac{\begin{array}{c}\text{there is no } \theta \text{ with}\\ \Delta \vdash \mathbf{t} \doteq \mathbf{s} \mid \theta \end{array} \quad \Delta \vdash \bot \;/\; S \;/\; F}{\Delta \vdash \mathbf{t} \doteq \mathbf{s} \;/\; S \;/\; F}$$

$$\frac{\Delta, x{:}\tau \vdash G \;/\; S \;/\; F \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \exists x{:}\tau.\, G \;/\; S \;/\; F}$$

$$\frac{\begin{array}{l}(\; p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o)\\ (\; \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}(\boldsymbol{\tau}/\boldsymbol{\alpha})\;)\\ \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p \in \Gamma \qquad \Delta \vdash G_p(\boldsymbol{\tau}/\boldsymbol{\alpha})(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F\end{array}}{\Delta \vdash p(\mathbf{t}) \;/\; S \;/\; F}$$

Figure 2: Operational Semantics Judgment $\Sigma; \Gamma; \Delta \vdash G \;/\; S \;/\; F$

Propositions $\Sigma; \Delta \vdash A : o$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \wedge B : o} \qquad \frac{}{\Delta \vdash \top : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \vee B : o} \qquad \frac{}{\Delta \vdash \bot : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \supset B : o} \qquad \frac{\Delta \vdash \mathbf{t} : \boldsymbol{\tau} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\tau}}{\Delta \vdash \mathbf{t} \doteq \mathbf{s} : o}$$

Terms $\Sigma; \Delta \vdash t : \tau,\ \Sigma; \Delta \vdash \mathbf{t} : \boldsymbol{\tau}$

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma \quad \Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash p(\mathbf{t}) : o}$$

$$\frac{\Delta, x{:}\tau \vdash A : o}{\Delta \vdash \forall x{:}\tau.\, A : o} \qquad \frac{\Delta, x{:}\tau \vdash A : o}{\Delta \vdash \exists x{:}\tau.\, A : o} \qquad \frac{\Delta, \alpha\ \mathit{type} \vdash A : o}{\Delta \vdash \forall \alpha.\, A : o}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \quad \Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

$$\frac{x{:}\tau \in \Delta}{\Delta \vdash x : \tau} \qquad \frac{\Delta \vdash t : \tau \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash (t, \mathbf{t}) : (\tau, \boldsymbol{\tau})} \qquad \frac{}{\Delta \vdash (\cdot) : (\cdot)}$$

Substitutions $\Sigma; \Delta \vdash \theta\ \mathit{subst}$

$$\frac{}{\Delta \vdash (\cdot)\ \mathit{subst}} \qquad \frac{\Delta \vdash \theta\ \mathit{subst} \quad x{:}\tau \in \Delta \quad \Delta \vdash t : \tau}{\Delta \vdash (\theta, t/x)\ \mathit{subst}}$$

Figure 3: Typing Judgments

Types $\Sigma; \Delta \vdash \tau\ \mathit{type},\ \Sigma; \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n$

$$\frac{a : \mathbf{type}_n \to \mathit{type} \in \Sigma \quad \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n}{\Delta \vdash a(\boldsymbol{\tau})\ \mathit{type}} \qquad \frac{\alpha\ \mathit{type} \in \Delta}{\Delta \vdash \alpha\ \mathit{type}}$$

$$\frac{}{\Delta \vdash (\cdot)\ \mathbf{type}_0} \qquad \frac{\Delta \vdash \tau\ \mathit{type} \quad \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n}{\Delta \vdash (\tau, \boldsymbol{\tau})\ \mathbf{type}_{n+1}}$$

Type Substitutions $\Sigma; \Delta \vdash \hat{\theta}\ \mathit{tsubst}$

$$\frac{}{\Delta \vdash (\cdot)\ \mathit{tsubst}} \qquad \frac{\Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \tau\ \mathit{type}}{\Delta \vdash (\hat{\theta}, \tau/\alpha)\ \mathit{tsubst}}$$

Signatures $\Sigma\ \mathit{sig}$

$$\frac{}{(\cdot)\ \mathit{sig}} \qquad \frac{\Sigma\ \mathit{sig}}{(\Sigma, a : \mathbf{type}_n \to \mathit{type})\ \mathit{sig}}$$

$$\frac{\Sigma\ \mathit{sig} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \boldsymbol{\sigma}\ \mathbf{type} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \tau\ \mathit{type}}{(\Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau)\ \mathit{sig}}$$

$$\frac{\Sigma\ \mathit{sig} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \boldsymbol{\sigma}\ \mathbf{type}}{(\Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o)\ \mathit{sig}}$$

Contexts $\Sigma; \Delta\ \mathit{ctx}$

$$\frac{}{(\cdot)\ \mathit{ctx}} \qquad \frac{\Delta\ \mathit{ctx}}{(\Delta, \alpha\ \mathit{type})\ \mathit{ctx}} \qquad \frac{\Delta\ \mathit{ctx} \quad \Delta \vdash \tau\ \mathit{type}}{(\Delta, x{:}\tau)\ \mathit{ctx}}$$

Programs $\Sigma; \Gamma\ \mathit{prog}$

$$\frac{}{(\cdot)\ \mathit{prog}} \qquad \frac{\Gamma\ \mathit{prog} \quad (p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o) \in \Sigma \quad \cdot \vdash \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p : o}{(\Gamma, \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p)\ \mathit{prog}}$$

States $\Sigma; \Gamma; \Delta \vdash G\ /\ S\ /\ F\ \mathit{state}$

$$\frac{\Sigma\ \mathit{sig} \quad \Gamma\ \mathit{prog} \quad \Delta\ \mathit{ctx} \quad \Delta \vdash G : o \quad \Delta \vdash S : o \quad \Delta \vdash F : o}{\Sigma; \Gamma; \Delta \vdash G\ /\ S\ /\ F\ \mathit{state}}$$

Figure 4: Well-Formedness Judgments

15-819K: Logic Programming

Lecture 11

Difference Lists

Frank Pfenning

October 3, 2006

In this lecture we look at programming techniques that are specific to logic programming, or at least significantly more easily expressed and reasoned about in logic programming than other paradigms. The first example is *difference lists*, which we use for a queue data structure, list reversal, an improvement of our earlier quicksort implementation, and a breadth-first logic programming engine that can be seen as the core of a theorem prover. We also introduce a program for peg solitaire as a prototype for state exploration. This will lead us towards considering imperative logic programming.

11.1 Functional Queues

We would like to implement a queue with operations to enqueue, dequeue, and test a queue for being empty. For illustration purposes we use a list of instructions `enq(`x`)` and `deq(`x`)`. Starting from an empty queue, we execute the instructions in the order given in the list. When the instruction list is empty we verify that the queue is also empty. Later we will use queues to implement a breadth-first logic programming interpreter.

First, a naive, and very inefficient implementation, where a queue is simply a list.

```
queue0(Is) :- q0(Is, []).

q0([enq(X)|Is], Q) :- append(Q, [X], Q2), q0(Is, Q2).
q0([deq(X)|Is], [X|Q]) :- q0(Is, Q).
q0([], []).
```

This is inefficient because of the repeaated calls to `append` which copy the queue.

In a more efficient functional implementation we instead maintain two lists, one the front of the list and one the back. We enqueue items on the back and dequeue them from the front. When the front is empty, we *reverse* the back and make it the new front.

```
queue1(Is) :- q1(Is, [], []).

q1([enq(X)|Is], F, B) :- q1(Is, F, [X|B]).
q1([deq(X)|Is], [X|F], B) :- q1(Is, F, B).
q1([deq(X)|Is], [], B) :- reverse(B,[X|F]), q1(Is,F,[]).
q1([], [], []).
```

Depending on the access patterns for queues, this can much more efficient since the first since the cost of the list reversal can be amortized over the enqueuing and dequeuing operations.

11.2 Queues as Difference Lists

The idea behind this implementation is that the a queue with elements $x_1, \ldots, x_n$ is represented as a pair `[`x_1`, ..., `x_n` | B] \ B`, where `B` is a logic variable. Here `\` is simply a constructor written in infix form to suggest list difference because the actual queue of elements for `F \ B` is the list `F` minus the tail `B`.

One may think of the variable `B` as a pointer to the end of the list, providing a means to add an element at the end in constant time (instead of calling `append` as in the very first implementation). Here is a first implementation using this idea:

```
queue(Is) :- q(Is, B\B).

q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
q([deq(X)|Is], [X|F]\B) :- q(Is, F\B).
q([], []\[]).
```

We consider is line by line, in each case considering the invariant:

> A queue $x_1, \ldots, x_n$ is represented by `[`x_1`, ..., `x_n` | B] \ B` for a logic variable `B`.

In the first clause

```
queue(Is) :- q(Is, B\B).
```

we see that the empty queue is represented as `B\B` for a logic variable `B`, which is an instance of the invariant for $n = 0$.

The second clause

```
q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
```

is trickier. A goal matching the head of this clause will have the form

?- q([enq(x_{n+1})|l], [x_1, ..., x_n | B0] \ B0).

for a term x_{n+1}, list l, terms $x_1, \ldots, x_n$ and variable `B0`. Unification will instantiate

X = x_{n+1}
Is = l
F = [x_1, ..., x_n, x_{n+1} | B]
B0 = [x_{n+1} | B]

where `B` is a fresh logic variable. Now the recursive call is

?- q(l, [x_1, ..., x_n, x_{n+1} | B1] \ B1).

satisfying our representation invariant.

The third clause

```
q([deq(X)|Is], [X|F]\B) :- q(Is, F\B).
```

looks straightforward, since we are just working on the front of the queue, removing its first element. However, there is a tricky issue when the queue is empty. In that case it has the form `B0 \ B0` for some *logic variable* `B0`, so it can actually unify with `[X|F]`. In that case, `B0 = [X|F]`, so the recursive call will be on `q(`l`, F \ [X|F])` which not only violates our invariant, but also unexpectedly allows us to remove an element from the empty queue!

The invariant will right itself once we enqueue another element that matches `X`. In other words, we have constructed a "negative" queue, borrowing against future elements that have not yet arrived. If this behavior is undesirable, it can be fixed in two ways: we can either add a counter as a third argument that tracks the number of elements in the queue, and then verify that the counter is positive before dequeuing and element. Or we can check if the queue is empty before dequeuing and fail explicitly in that case.

Let us consider the last clause.

```
q([], []\[]).
```

This checks that the queue is empty by unifying it with `[]\[]`. From the invariant we can see that it succeeds exactly if the queue is empty (neither positive nor negative, if borrowing is allowed).

When we started with the empty queue, we used the phrase `B\B` for a logic variable `B` to represent the empty queue. Logically, this is equivalent to checking unifiability with `[]\[]`, but operationally this does not work because of the lack of occurs-check in Prolog. A non-empty queue such as `[`x_1`|B0] \ B0` will incorrectly "unify" with `B1\B1` with `B1` being instantiated to a circular term `B1 = [`x_1`|B1]`.

To complete this example, we show the version that prevents negative queues by testing if the front is unifiable with `[]` before proceeding with a dequeue operation.

```
queue(Is) :- q(Is, B\B).

q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
q([deq(X)|Is], F\B) :-
        F = [] -> fail ; F = [X|F1], q(Is, F1\B).
q([], []\[]).
```

11.3 Other Uses of Difference Lists

In the queue example, it was important that the tail of the list is always a logic variable. There are other uses of difference list where this is not required. As a simple example consider `reverse`. In its naive formulation it overuses `append`, as in the naive formulation of queues.

```
naive_reverse([X|Xs], Zs) :-
        naive_reverse(Xs, Ys),
        append(Ys, [X], Zs).
naive_reverse([], []).
```

To make this more efficient, we use a difference list as the second argument.

```
reverse(Xs, Ys) :- rev(Xs, Ys\[]).

rev([X|Xs], Ys\Zs) :- rev(Xs, Ys\[X|Zs]).
rev([], Ys\Ys).
```

This time, the front of the difference list is a logic variable, to be filled in when the input list is empty.

Even though this program is certainly correctly interpreted using list difference, the use here corresponds straightforwardly to the idea of *accumulators* in functional programming: In `rev(Xs, Ys\Zs)`, `Zs` accumulates the reverse list and eventually returns it in `Ys`.

Seasoned Prolog hackers will often break up an argument which is a difference list into two top-level arguments for efficiency reasons. So the `reverse` code above might actually look like

```
reverse(Xs, Ys) :- rev(Xs, Ys, []).

rev([X|Xs], Ys, Zs) :- rev(Xs, Ys, [X|Zs]).
rev([], Ys, Ys).
```

where the connection to difference lists is harder to recognize.

Another useful example of difference lists is in quicksort from Lecture 2, omitting here the code for `partition/4`. We construct two lists, `Ys1` and `Ys2` and append them, copying `Ys1` again.

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

Instead, we can use a difference list.

```
quicksort(Xs, Ys) :-
        qsort(Xs, Ys\[]).

qsort([], Ys\Ys).
qsort([X0|Xs], Ys\Zs) :-
        partition(Xs, X0, Ls, Gs),
        qsort(Gs, Ys2\Zs),
        qsort(Ls, Ys\[X0|Ys2]).
```

In this instance of difference lists, it may be helpful to think of

`qsort(`Xs`,` $Ys\backslash Zs$`)`

as adding the sorted version of Xs to the front of Zs to obtain Ys. Then, indeed, the result of subtracting Zs from Ys is the sorted version of Xs. In order to see this most directly, we have swapped the two recursive calls to `qsort` so that the tail of the difference list is always ground on invocation.

11.4 A Breadth-First Logic Programming Interpreter

With the ideas of queues outlined above, we can easily construct a breadth-first interpreter for logic programs. Breadth-first search consumes a lot of space, and it is very difficult for the programmer to obtain a good model of program efficiency, so this is best thought of as a naive, first attempt at a theorem prover that can find proofs even where the interpreter would loop.

The code can be found on the course website.[1] It is obtained in a pretty simple way from the depth-first interpreter, by replacing the failure continuation F by a pair $F_1 \backslash F_2$, where F_2 is always a logic variable that occurs at the tail of F_1. While F_1 is not literally a list, it should be easy to see what this means.

We show here only four interesting clauses. First three that are directly concerned with the failure continuation.

```
% prove(G, Gamma, S, F1\F2, N, J)
% Gamma |- G / S / FQ, N is next free variable
% J = success or failure
...
prove(bot, _, _, bot\bot, _, J) :- !, J = failure.
prove(bot, Gamma, _, or(and(G2,S),F1)\F2, N, J) :-
        prove(G2, Gamma, S, F1\F2, N, J).
prove(or(G1,G2), Gamma, S, F1\or(and(G2,S),F2), N, J) :-
        prove(G1, Gamma, S, F1\F2, N, J).
```

The first clause here needs to commit so that the second clause cannot borrow against the future. It should match only if there is an element in the queue.

In order to make this prover complete, we need to cede control immediately after an atomic predicate is invoked. Otherwise predicates such as

```
diverge :- diverge ; true.
```

would still not terminate since alternatives, even though queued rather than stacked, would never be considered. The code for this case looks like

[1] `http://www.cs.cmu.edu/~fp/courses/lp/code/11-diff/meta.pl`

```
prove(app(Pred, Ts), Gamma, S, FQ, N, J) :-
        ...
        prove(or(bot,GTheta), Gamma, S, FQ, N, J).
```

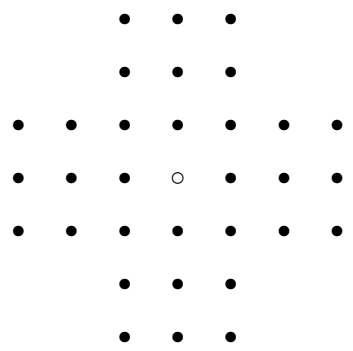
where `GTheta` will be the body of the definition of `Pred`, with arguments `Ts` substituted for the argument variables. We assume here that the program is already in residuated form, as is necessary for this semantics to be accurate.

In the next step we will queue up `GTheta` on the failure continuation and then fail while trying to prove `bot`. Another alternative, suspended earlier, will then be removed from the front of the queue and its proof attempted.[2]

11.5 State Exploration

We now switch to a different category of programs, broadly categorized as exploring state. Game playing programs are in this class, as are puzzle solving programs. One feature of logic programming we hope to exploit is the backtracking nature of the operational semantics.

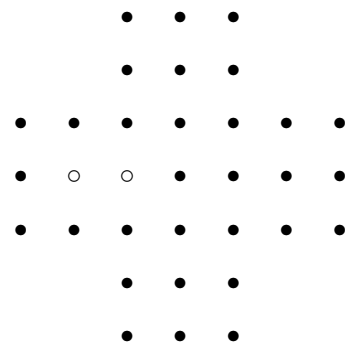
The example we use is *peg solitaire*. We are given a board of the form



where a solid circle • is a hole filled with a peg, while a ○ hollow circle represents an empty hole. In each move, a peg can jump over an adjacent one (right, down, left, or up), if the hole behind is empty. The peg that is jumped over is removed from the board. For example, in the initial position shown above there are four possible moves, all ending up in the center. If

[2] I have no proof that this really is complete—I would be interested in thoughts on the issue.

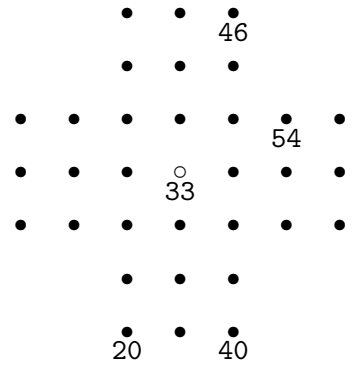
we take the possible jump to the right, we would be in the position



The objective is continue jumps until only one peg is left over.

This puzzle has been extensively analyzed (see, for example, the Wikipedia article on the subject). Our logic programming implementation will to inefficient to solve the problem by brute force, but it is nonetheless an illustrative example.[3]

We introduce a unique name for every place on the board by using a integer coordinate address and concatenating the two digits starting with `00` at the lower-left hand corner which is unoccupied, proceeding to `66` in the upper right-hand corner. Some place names are drawn in the following diagram.



Now the current state of the board during the search for a solution is represented by a list containing `peg(`ij`)` if there is a peg at location ij and

[3]This program was written by committee in real-time during lecture. Another somewhat more efficient version can be found with code that accompanies the lecture at `http://www.cs.cmu.edu/~fp/courses/lp/code/11-diff/`.

`hole(`ij`)` if the location ij is an empty hole. We have a prediate `init(`S_0`)` which holds for the initial state S_0.

```
init([
                peg(20),peg(30),peg(40),
                peg(21),peg(31),peg(41),
peg(02),peg(12),peg(22),peg(32),peg(42),peg(52),peg(62),
peg(03),peg(13),peg(23),hole(33),peg(43),peg(53),peg(63),
peg(04),peg(14),peg(24),peg(34),peg(44),peg(54),peg(64),
                peg(25),peg(35),peg(45),
                peg(26),peg(36),peg(46)
]).
```

We also have a predicate `between/3` which holds between three places A, B, and C whenever there is a possible jump to the right or up. This means that if `between(`C`,`B`,`A`)` is true then a jump left or up from A to C is possible. We show a few cases in the definition of `between`.

```
between(20,30,40).
between(20,21,22).
between(30,31,32).
between(40,41,42).
...
```

There are 38 such clauses altogether.

Next we have a predicate to flip a peg to a hole in a given state returning the new state.

```
swap([peg(A)|State], peg(A), [hole(A)|State]).
swap([hole(A)|State], hole(A), [peg(A)|State]).
swap([Place|State1], Place0, [Place|State2]) :-
        swap(State1, Place0, State2).
```

This fails if the requested place is not a peg or hole, respectively.

In order to make a single move, we find all candidate triples A, B, or C using the `between` relation and then swap the two pegs and hole to be two holes and a peg.

```
move1(State1,State4) :-
        ( between(A,B,C) ; between(C,B,A) ),
        swap(State1, peg(A), State2),
        swap(State2, peg(B), State3),
        swap(State3, hole(C), State4).
```

To see if we can make n moves from a given state we make one move and then see if we can make $n - 1$ moves from the resulting state. If this fails, we backtrack, trying another move.

```
moves(0, _).
moves(N, State1) :-
        N > 0,
        move1(State1, State2),
        N1 is N-1,
        moves(N1, State2).
```

Finally, to solve the puzzle we have to make n moves from the initial state. To have a full solution, we would need $n = 31$, since we start with 32 pegs so making 31 moves will win.

```
solve(N) :-
        init(State0),
        moves(N, State0).
```

Since in practice we cannot solve the puzzle this way, it is interesting to see how many sequences of moves of length n are possible from the initial state. For example, there are 4 possible sequences of a single move and, 12 possible sequences of two moves, and 221072 sequences of seven moves.

There we encounter a difficulty, namely that we cannot maintain any information about the number of solutions upon backtracking in pure Prolog (even though as you have seen in a homework assignments, it is easy to count the number of solutions in the meta-interpreter).

This inability to preserve information is fundamental, so Prolog implementations offer several ways to circumvent it. One class of solutions is represented by `findall` and related predicates which can collect the solutions to a query in a list. A second possibility is to use `assert` and `retract` to change the program destructively while it is running—a decidedly non-logical solution. Final, modern Prolog implementations offer global variables that can be assigned to and incremented in a way that survives backtracking.

We briefly show the third solution in GNU Prolog. A global variable is addressed by an atom, here `count`. We can assign to it with `g_assign/2`, increment it with `g_inc/1` and read its value with `g_read/2`. The idea is to let a call to `solve` succeed, increment a global variable `count`, then fail and backtrack into `solve` to find an another solution, etc., until there are no further solutions.

```
test(N) :-
        g_assign(count, 0),
        solve(N),
        g_inc(count),
        fail.
test(N) :-
        g_read(count, K),
        format("At ~p, ~p solutions\n", [N,K]).
```

This works, because the effect of incrementing `count` remains, even when backtracking fails past it.

Although global variables are clearly non-logical, their operational semantics is not so difficult to specify (see Exercise 11.4).

This example reveals imperative or potentially imperative operations on several levels. In the next lecture we will explore one of them.

11.6 Historical Notes

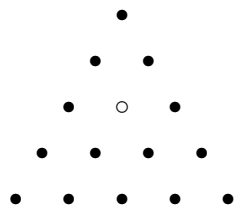
Okasaki has given a perceptive analysis of purely functional queues [2]. According to Sterling and Shapiro [3], difference lists have been Prolog folklore since the early days, with a first published description in a paper by Clark and Tärnlund [1].

11.7 Exercises

Exercise 11.1 *Reconsider the Dutch national flag problem introduced in Exercise 2.4 and give an efficient solution using difference lists.*

Exercise 11.2 *Think of another interesting application of difference lists or related incomplete data structures and write and explain your implementation.*

Exercise 11.3 *Give a Prolog implementation of the following triangular version of peg solitaire*



where jumps can be made in 6 directions: east, northeast, northwest, west, southwest, and southeast (but not directly north or south). Use your program to determine the number of solutions (you may count symmetric ones), and in which locations the only remaining peg may end up in. Also, what is the maximal number of pegs that may be left on the board without any possible further moves?

Exercise 11.4 *Give an extension of the operational semantics with goal stacks, failure continuations, and explicit unification to model global variables (named by constants) which can be assigned, incremented, decremented, and read. Glean their intended meaning from the use in the* `test/1` *predicate for peg solitaire.*

You may assume that the values assigned to global variables in this manner are ground and remain unaffected by backtracking.

11.8 References

[1] K. L. Clark and S.-A. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Proceedings of the IFIP Congress*, pages 939–944, Toronto, Canada, 1977. North Holland.

[2] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[3] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2nd edition edition, 1994.

15-819K: Logic Programming

Lecture 12

Linear Logic

Frank Pfenning

October 5, 2006

In this lecture we will rewrite the program for peg solitaire in a way that treats state logically, rather than as an explicit data structure. In order to allow this we need to generalize the logic to handle state intrinsically, something provided by *linear logic*. We provide an introduction to linear logic as a *sequent calculus*, which generalizes our previous way of specifying truth. The sequent calculus is a bit too general to allow an immediate operational interpretation to obtain a logic programming language, so we postpone this step to the next lecture.

12.1 State-Passing Style

Let us reexamine the central part of the program for peg solitaire from the last lecture. The first predicate `moves` passes state downward in the program.

```
moves(0, _).
moves(N, State1) :-
        N > 0,
        move1(State1, State2),
        N1 is N-1,
        moves(N1, State2).
```

The second and third predicates, `move1` and `swap` pass state from an input argument (the first) to an output argument (the last).

```
move1(State1, State4) :-
        ( between(A,B,C) ; between(C,B,A) ),
        swap(State1, peg(A), State2),
        swap(State2, peg(B), State3),
        swap(State3, hole(C), State4).

swap([peg(A)|State], peg(A), [hole(A)|State]).
swap([hole(A)|State], hole(A), [peg(A)|State]).
swap([Place|State1], Place0, [Place|State2]) :-
        swap(State1, Place0, State2).
```

This pattern of code is called *state-passing* or *store-passing*. State-passing style is a common pattern in logic programs of a certain kind, specifically state exploration as in puzzles and games or modeling concurrent or distributed systems.

We investigate in this and the next lecture how to eliminate explicit state passing from such programs in favor of logical primitives.

In functional programming, the related *store-passing* style usually arises in the opposite way: if we want to turn a functional program that uses mutable storage into a pure functional program we can pass the store around as an explicit argument.

12.2 State-Dependent Truth

In our program, state is represented as a list of items `peg(`ij`)` and `hole(`ij`)` for locations ij. Stepping back from this particular representation, it is easy to interpret `peg` and `hole` as *predicates*, and `peg(`ij`)` and `hole(`ij`)` as *propositions*. For example, we say the proposition `peg(`ij`)` is true if there is a peg in location ij on the board.

What makes this somewhat unusual, from the perspective of the logic we have considered so far, is that the notion of truth depends on the state. In some states, `peg(`ij`)` is true, in some it is false. In fact, the state of the board is completely characterized by the `peg` and `hole` propositions.

In mathematical logic, truth is normally invariant and does depend on state. This is because the mathematical objects we deal with, such as natural numbers, are themselved invariant and considered universal. In philosophical logic, however, the concept of truth depending on the state of the world is central and has been investigated under the name *modal logic*, of which *temporal logic* is a particular branch. In these logics truth explicitly depends on the state of the world, and the separate concept of necessary

truth captures those properties that are state invariant. However, neither modal nor temporal logic is particularly appropriate for our problem domain. As an example, let us consider how we would specify a legal state transition, using the next-time operator $\bigcirc$. We might write

$$\begin{array}{c}\mathsf{between}(A,B,C) \wedge \mathsf{peg}(A) \wedge \mathsf{peg}(B) \wedge \mathsf{hole}(C) \\ \supset \bigcirc(\mathsf{hole}(A) \wedge \mathsf{hole}(B) \wedge \mathsf{hole}(C)).\end{array}$$

Unfortunately, unless we specify something else, the *only* thing we know about the next state is $\mathsf{hole}(A) \wedge \mathsf{hole}(B) \wedge \mathsf{hole}(C)$. What we would really like to say is that all other propositions regarding locations besides A, B, and C remain unchanged.

This kind of circumscription is awkward, defeating the purpose of obtaining a higher-level and more elegant formulation of our example and similar state-passing code. Moreoever, when we add more predicates then the move specification must also change to carry these over unchanged. In artificial intelligence this is called the *frame problem*. In the next section we show an elegant and completely logical solution to this problem.

12.3 Linear Logic

Linear logic has been described as a logic of state or a resource-aware logic.[1] Formally, it arises from complementing the usual notion of logical assumption with so-called *linear assumptions* or *linear hypotheses*. Unlike traditional assumptions which may be used many times in a proof, linear assumptions must be used *exactly once* during a proof. Linear assumptions then become (consumable) *resources* in the course of a proof.

This generalization of the usual mathematical standpoint may seem slight, but as we will see it is quite expressive. We write

$$A_1\ \mathit{res}, \ldots, A_n\ \mathit{res} \Vdash C\ \mathit{true}$$

for a linear hypothetical judgment with resources $A_1, \ldots, A_n$ and goal C. If we can prove this, it means that we can achieve that C is true, given resources A_1 through A_n. Here, all A_i and C are propositions.[2] The version of linear logic defined by this judgment is called *intuitionistic linear logic*,

[1] The term *linear* is connected to its use in algebra, but the connection is not easy to explain. For this lecture just think of "*linear*" as denoting "*must be used exactly once*".

[2] In the end it will turn out that $A\ \mathit{res}$ and $A\ \mathit{true}$ are interchangeable in that we can go from each one to the other. At this point, however, we do not know this yet, so the judgment we make about our resources is not that they are true, but that they are given resources.

sometimes contrasted with *classical linear logic* in which the sequent calculus has multiple conclusions. While it is possible to develop classical linear logic programming it is more difficult to understand and use.

Hidden in the judgment are other assumptions, usually abbreviated as Γ, which can be used arbitrarily often (including not at all), and are therefore called the *unrestricted assumptions*. If we need to make them explicit in a rule we will write

$$\Gamma; \Delta \Vdash C \ true$$

where Δ abbreviates the resources. As in our development so far, unrestricted assumption are fixed and are carried through from every conclusion to all premisses. Eventually, we will want to generalize this, but not quite yet.

The first rule of linear logic is that if we have a resource P we can achieve goal P, where P is an atomic proposition. It will be a consequence of our definitions that this will be true for arbitrary propositions A, but we need it as a rule only for the atomic case, where the structure of the propositions can not be broken down further.

$$\frac{}{P \ res \Vdash P \ true} \ \mathrm{id}$$

We call this the *identity rule*, it is also sometimes called the *init* rule, and the sequent $P \Vdash P$ is called an *initial sequent*.

12.4 Connectives of Linear Logic

One of the curious phenomena of linear logic is that the ordinary connectives multiply. This is because the presence of linear assumptions allows us to make distinctions we ordinarily could not. The first example of this kind is conjunction. It turns out that linear logic possesses two forms of conjunction.

Simultaneous Conjunction ($A \otimes B$). A simultaneous conjunction $A \otimes B$ is true if we can achieve both A and B in the same state. This means we have to subdivide our resources, devoting some of them to achieve A and the others to achieve B.

$$\frac{\Delta = (\Delta_A, \Delta_B) \quad \Delta_A \Vdash A \quad \Delta_B \Vdash B}{\Delta \Vdash A \otimes B} \otimes R$$

The order of linear assumptions is irrelevant, so in $\Delta = (\Delta_A, \Delta_B)$ the comma denotes the multi-set union. In other words, every occurrence of a proposition in Δ will end up in exactly one of Δ_A and Δ_B.

If we name the initial state of peg solitaire Δ_0, then we have $\Delta_0 \Vdash \mathsf{peg}(33) \otimes \mathsf{hole}(03) \otimes \ldots$ for some "..." because we can achieve a state with a peg at location 33 and hole at location 03. On the other hand, we cannot prove $\Delta_0 \Vdash \mathsf{peg}(33) \otimes \mathsf{hole}(33) \otimes \ldots$ because we cannot have a peg and an empty hole at location 33 in the same state. We will make the ellipsis "..." precise below as consumptive truth $\top$.

In a linear sequent calculus, the right rules shows when we can conclude a proposition. The left rule shows how we can use a resource. In this case, the resource $A \otimes B$ means that we have A and B simultaneously, so the left rule reads

$$\frac{\Delta, A\ res, B\ res \Vdash C\ true}{\Delta, A \otimes B\ res \Vdash C\ true}\ \otimes L.$$

In comparison to the focusing judgment we used to explain the logical semantics of pure Prolog programs, the left rules are not restricted to continuously decompose a single proposition until an atomic form is reached. Instead, various applicable left rules that operate on different assumptions can be freely interleaved. We consider the restriction to focusing in the next lecture.

Alternative Conjunction ($A \mathbin{\&} B$). An alternative conjunction is true if we can achieve both conjuncts, separately, with the current resources. This means if we have a linear assumption $A \mathbin{\&} B$ we have to make a choice: either we use A or we use B, but we cannot use them both.

$$\frac{\Delta \Vdash A\ true \quad \Delta \Vdash B\ true}{\Delta \Vdash A \mathbin{\&} B\ true}\ \&R$$

$$\frac{\Delta, A\ res \Vdash C\ true}{\Delta, A \mathbin{\&} B\ res \Vdash C\ true}\ \&L_1 \qquad \frac{\Delta, B\ res \Vdash C\ true}{\Delta, A \mathbin{\&} B\ res \Vdash C\ true}\ \&L_2$$

It looks like the right rule duplicates the assumptions, but this does not violate linearity because in a use of the assumption $A \mathbin{\&} B\ res$ we have to commit to one or the other.

Returning to the solitaire example, we have $\Delta_0 \Vdash \mathsf{peg}(33) \otimes \mathsf{hole}(03) \otimes \ldots$ and we also have $\Delta_0 \Vdash \mathsf{hole}(33) \otimes \mathsf{hole}(03) \otimes \ldots$ because we can certainly reach states with these properties. However, we cannot reach a single state with both of these, because the two properties of location 33 clash. If we

want to express that both are reachable, we can form their alternative conjunction

$$\Delta_0 \Vdash (\mathsf{peg}(33) \otimes \mathsf{hole}(03) \otimes \ldots) \,\&\, (\mathsf{hole}(33) \otimes \mathsf{hole}(03) \ldots).$$

Consumptive Truth ($\top$). We have seen two forms of conjunction, which are distinguished because of their resource behavior. There are also two truth constants, which correspond to zero-ary conjunctions. The first is *consumptive truth* $\top$. A proof of it consumes all current resources. As such we can extract no information from its presence as an assumption.

$$\frac{}{\Delta \Vdash \top\ \mathit{true}}\ \top R \qquad\qquad \begin{array}{c}\text{no } \top L \text{ rule} \\ \Delta, \top\ \mathit{res} \Vdash C\ \mathit{true}\end{array}$$

Consumptive truth is important in applications where there is an aspect of the state we do not care about, because of the stipulation of linear logic that every linear assumption must be used *exactly once*. In the examples above so far we cared about only two locations, 33 and 03. The state will have a linear assumption for every location, which means we can *not* prove, for example, $\Delta_0 \Vdash \mathsf{peg}(33) \otimes \mathsf{hole}(03)$. However, we *can* prove $\Delta_0 \Vdash \mathsf{peg}(33) \otimes \mathsf{hole}(03) \otimes \top$, because the consumptive truth matches the remaining state.

Consumptive truth is the unit of alternative conjunction in that $A \,\&\, \top$ is equivalent to A.

Empty Truth (1). The other form of truth holds only if there are no resources. If we have this as a linear hypothesis we can transform it into the empty set of resources.

$$\frac{\Delta = (\cdot)}{\Delta \Vdash \mathbf{1}\ \mathit{true}}\ \mathbf{1}R \qquad\qquad \frac{\Delta \Vdash C\ \mathit{true}}{\Delta, \mathbf{1}\ \mathit{res} \Vdash C\ \mathit{true}}\ \mathbf{1}L$$

Empty truth can be useful to dispose explicitly of specific resources.

Linear Implication ($A \multimap B$). A linear implication $A \multimap B$ is true if we can achieve B given resource A.

$$\frac{\Delta, A\ \mathit{res} \Vdash B\ \mathit{true}}{\Delta \Vdash A \multimap B\ \mathit{true}}\ \multimap R$$

Conversely, if we have $A \multimap B$ as a resource, it means that we could transform the resource A into the resource B. We capture this in the following left rule:

$$\frac{\Delta = (\Delta_A, \Delta_B) \quad \Delta_A \Vdash A\ true \quad \Delta_B, B\ res \Vdash C\ true}{\Delta, A \multimap B\ res \Vdash C\ true} \multimap L.$$

An assumption $A \multimap B$ therefore represents a means to transition from a state with A to a state with B.

Unrestricted Assumptions Γ. The left rule for linear implication points at a problem: the linear implication is itself linear and therefore consumed in the application of that rule. If we want to specify via a linear logic program how state may change, we will need to reuse the clauses over and over again. This can be accomplished by a *copy* rule which takes an unrestricted assumption and makes a linear copy of it. It is actually very much like the focusing rule in an earlier system.

$$\frac{A\ ures \in \Gamma \quad \Gamma; \Delta, A\ res \Vdash C\ true}{\Gamma; \Delta \Vdash C\ true}\ \text{copy}$$

We label the unrestricted assumptions as unrestricted resources, $A\ ures$. In the logic programming interpretation, the whole program will end up in Γ as unrestricted assumptions, since the program clauses can be used arbitrarily often during a computation.

Resource Independence ($!A$). The proposition $!A$ is true if we can prove A without using any resources. This means we can produce as many copies of A as we need (since it costs nothing) and a linear resource $!A$ licenses us to make the unrestricted assumption A.

$$\frac{\Gamma; \cdot \Vdash A\ true}{\Gamma; \cdot \Vdash !A\ true}\ !R \qquad \frac{(\Gamma, A\ ures); \Delta \Vdash C\ true}{\Gamma; \Delta, !A\ res \Vdash C\ true}\ !L$$

Disjunction ($A \oplus B$). The familiar conjunction from logic was split into two connectives in linear logic: the simultaneous and the alternative conjunction. Disjunction does not split the same way unless we introduce an explicit judgment for falsehood (which we will not pursue). The goal $A \oplus B$ can be achieved if we can achieve either A or B.

$$\frac{\Delta \Vdash A\ true}{\Delta \Vdash A \oplus B\ true} \oplus R_1 \qquad \frac{\Delta \Vdash B\ true}{\Delta \Vdash A \oplus B\ true} \oplus R_2$$

Conversely, if we are given $A \oplus B$ as a resource, we do not know which of the two is true, so we have to account for both eventualities. Our proof splits into cases, and we have to show that we can achieve our goal in either case.

$$\frac{\Delta, A\ res \Vdash C\ true \quad \Delta, B\ res \Vdash C\ true}{\Delta, A \oplus B\ res \Vdash C\ true}\ \oplus L$$

Again, it might appear as if linearity is violated due to the duplication of Δ and even C. However, only one of A or B will be true, so only one part of the plan represented by the two premisses really applies, preserving linearity.

Falsehood (0). There is no way to prove falsehood $\mathbf{0}$, so there is no right rule for it. On the other hand, if we have $\mathbf{0}$ as an assumption we know we are really in an impossible state so we are permitted to succeed.

$$\begin{array}{l}\text{no } \mathbf{0}R \text{ rule} \\ \Delta \Vdash \mathbf{0}\ true\end{array} \qquad \frac{}{\Delta, \mathbf{0}\ res \Vdash C\ true}\ \mathbf{0}L$$

We can also formally think of falsehood as a disjunction between zero alternatives and arrive at the same rule.

12.5 Resource Management

The connectives of linear logic are generally classified into *multiplicative*, *additive*, and *exponential*.[3]

The multiplicative connectives, when their rules are read from conclusion to the premisses, split their resources between the premisses. The connectives $\otimes$, $\mathbf{1}$, and $\multimap$ have this flavor.

The additive connectives, when their rules are read from conclusion to premisses, propagate their resources to all premisses. The connectives $\&$, $\top$, $\oplus$, and $\mathbf{0}$ have this flavor.

The exponential connectives mediate the boundary between linear and non-linear reasoning. The connective ! has this flavor.

During proof search (and therefore in the logic programming setting), a significant question is how to handle the resources. It is clearly impractical, for example, in the rule

$$\frac{\Delta = (\Delta_A, \Delta_B) \quad \Delta_A \Vdash A\ true \quad \Delta_B \Vdash B\ true}{\Delta \Vdash A \otimes B\ true}\ \otimes R$$

[3] Again, we will not try to explain the mathematical origins of this terminology.

to simply enumerate all possibilities and try to prove A and B in each combination until one is found that works for both.

Instead, we pass in all resources Δ into the first subgoal A and keep track which resources are consumed. We then pass the remaining ones to the proof of B. Of course, if B fails we may have to find another proof of A which consumes a different set of resources and then retry B, and so on. In the logic programming setting this is certainly an issue the programmer has to be aware of, just as the programmer has to know which subgoal is solved first, or which clause is tried first.

We will return to this question in the next lecture where we will make resource-passing explicit in the operational semantics.

12.6 Peg Solitaire, Linearly

We now return to the peg solitaire example. We coule like to rewrite the `moves` predicate from a state-passing `moves(`n`,`s`)` to just `moves(`n`)`, where the state s is actually encoded in the linear context Δ. That is, we consider the sitation

$$\Delta \Vdash \mathsf{moves}(n)$$

where Δ contains a linear assumption $\mathsf{peg}(ij)$ when there is a peg in location ij and $\mathsf{hole}(ij)$ if there is an empty hole in location ij.

The first clause,

```
moves(0, _).
```

is translated to

```
moves(0) ⟜ ⊤.
```

where $\multimap$ is reverse linear implication. The $\top$ here is necessary to consume the state (which, in this case, we don't care about).

The second clause for `moves`

```
moves(N, State1) :-
        N > 0,
        move1(State1, State2),
        N1 is N-1,
        moves(N1, State2).
```

as well as the auxiliary predicates `move1` and `swap` are replaced by just one clause in the definition of `moves`.

```
moves(N) ⟜
        N > 0 ⊗ N1 is N-1 ⊗
        ( between(A,B,C) ⊕ between(C,B,A) ) ⊗
        peg(A) ⊗ peg(B) ⊗ hole(C) ⊗
        (hole(A) ⊗ hole(B) ⊗ peg(C) ⊸ moves(N1)).
```

Operationally, we first compute $n-1$ and then find a triple A, B, C such that B is between A and C. These operations are state independent, although the clause does not indicate that.

At this point we determine if there are pegs at A and B and a hole at C. If this is not the case, we fail and backtrack; if it is we remove these three assumptions from the linear context (they are consumed!) and assume instead $\mathsf{hole}(A)$, $\mathsf{hole}(B)$, and $\mathsf{peg}(C)$ before calling moves recursively with $n - 1$. At this point this is the only outstanding subgoal, and the state has changed by jumping A over B into C, as specified.

Observe how linearity and the intrinsic handling of state let's us replace a lot of code for state management with one short clause.

12.7 Historical Notes

Linear logic in a slightly different form than we present here is due to Girard [2]. He insisted on a classical negation in his formulation, which can get in the way of an elegant logic programming formulation. The judgmental presentation we use here was developed for several courses on *Linear Logic* [3] at CMU. Some additional connectives, and some interesting connections between the two formulations in linear logic are developed by Chang, Chaudhuri and Pfenning [1]. We'll provide some references on linear logic programming in the next lecture.

12.8 Exercises

Exercise 12.1 *Prove that $A\ res \Vdash A\ true$ for any proposition A.*

Exercise 12.2 *For each of the following purely linear entailments, give a proof that they hold or demonstrate that they do not hold because there is no deduction in our system. You do not need to prove formally that no deduction exists.*

i. $A \,\&\, (B \oplus C) \Vdash (A \,\&\, B) \oplus (A \,\&\, C)$

ii. $A \otimes (B \oplus C) \Vdash (A \otimes B) \oplus (A \otimes C)$

iii. $A \oplus (B \,\&\, C) \Vdash (A \oplus B) \,\&\, (A \oplus C)$

iv. $A \oplus (B \otimes C) \Vdash (A \oplus B) \otimes (A \oplus C)$

Exercise 12.3 *Repeat Exercise 12.2 by checking the reverse linear entailments.*

Exercise 12.4 *For each of the following purely linear entailments, give a proof that they hold or demonstrate that they do not hold because there is no deduction in our system. You do not need to prove formally that no deduction exists.*

i. $A \multimap (B \multimap C) \Vdash (A \otimes B) \multimap C$

ii. $(A \otimes B) \multimap C \Vdash A \multimap (B \multimap C)$

iii. $A \multimap (B \mathbin{\&} C) \Vdash (A \multimap B) \mathbin{\&} (A \multimap C)$

iv. $(A \multimap B) \mathbin{\&} (A \multimap C) \Vdash A \multimap (B \mathbin{\&} C)$

v. $(A \oplus B) \multimap C \Vdash (A \multimap C) \mathbin{\&} (A \multimap C)$

vi. $(A \multimap C) \mathbin{\&} (A \multimap C) \Vdash (A \oplus B) \multimap C$

Exercise 12.5 *For each of the following purely linear entailments, give a proof that they hold or demonstrate that they do not hold because there is no deduction in our system. You do not need to prove formally that no deduction exists.*

i. $C \Vdash \mathbf{1} \multimap C$

ii. $\mathbf{1} \multimap C \Vdash C$

iii. $A \multimap \top \Vdash \top$

iv. $\top \Vdash A \multimap \top$

v. $\mathbf{0} \multimap C \Vdash \top$

vi. $\top \Vdash \mathbf{0} \multimap C$

Exercise 12.6 *For each of the following purely linear entailments, give a proof that they hold or demonstrate that they do not hold because there is no deduction in our system. You do not need to prove formally that no deduction exists.*

i. $!(A \otimes B) \Vdash !A \otimes !B$

ii. $!A \otimes !B \Vdash !(A \otimes B)$

iii. $!(A \mathbin{\&} B) \Vdash !A \otimes !B$

iv. $!A \otimes !B \Vdash !(A \,\&\, B)$

v. $!\top \Vdash \mathbf{1}$

vi. $\mathbf{1} \Vdash !\top$

vii. $!\mathbf{1} \Vdash \top$

viii. $\top \Vdash !\mathbf{1}$

ix. $!!A \Vdash !A$

x. $!A \Vdash !!A$

12.9 References

[1] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, December 2003.

[2] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[3] Frank Pfenning. Linear logic. Lecture Notes for a course at Carnegie Mellon University, 1995. Revised 1998, 2001.

12.10 Appendix: Summary of Intuitionistic Linear Logic

In the rules below, we show the unrestricted assumptions Γ only where affected by the rule. In all other rules it is propagated unchanged from the conclusion to all the premisses. Also recall that the order of hypotheses is irrelevant, and Δ_A, Δ_B stands for the multiset union of two collections of linear assumptions.

Judgmental Rules

$$\frac{}{P\ res \Vdash P\ true}\ \text{id} \qquad \frac{A\ ures \in \Gamma \quad \Gamma; \Delta, A\ res \Vdash C\ true}{\Gamma; \Delta \Vdash C\ true}\ \text{copy}$$

Multiplicative Connectives

$$\frac{\Delta_A \Vdash A \quad \Delta_B \Vdash B}{\Delta_A, \Delta_B \Vdash A \otimes B}\ \otimes R \qquad \frac{\Delta, A\ res, B\ res \Vdash C\ true}{\Delta, A \otimes B\ res \Vdash C\ true}\ \otimes L$$

$$\frac{}{\cdot \Vdash \mathbf{1}\ true}\ \mathbf{1}R \qquad \frac{\Delta \Vdash C\ true}{\Delta, \mathbf{1}\ res \Vdash C\ true}\ \mathbf{1}L$$

$$\frac{\Delta, A\ res \Vdash B\ true}{\Delta \Vdash A \multimap B\ true}\ \multimap R \qquad \frac{\Delta_A \Vdash A\ true \quad \Delta_B, B\ res \Vdash C\ true}{\Delta_A, \Delta_B, A \multimap B\ res \Vdash C\ true}\ \multimap L$$

Additive Connectives

$$\frac{\Delta \Vdash A\ true \quad \Delta \Vdash B\ true}{\Delta \Vdash A \,\&\, B\ true}\ \&R \qquad \frac{\Delta, A\ res \Vdash C\ true}{\Delta, A \,\&\, B\ res \Vdash C\ true}\ \&L_1 \qquad \frac{\Delta, B\ res \Vdash C\ true}{\Delta, A \,\&\, B\ res \Vdash C\ true}\ \&L_2$$

$$\frac{}{\Delta \Vdash \top\ true}\ \top R \qquad \text{no } \top L \text{ rule}$$

$$\frac{\Delta \Vdash A\ true}{\Delta \Vdash A \oplus B\ true}\ \oplus R_1 \qquad \frac{\Delta, A\ res \Vdash C\ true \quad \Delta, B\ res \Vdash C\ true}{\Delta, A \oplus B\ res \Vdash C\ true}\ \oplus L \qquad \frac{\Delta \Vdash B\ true}{\Delta \Vdash A \oplus B\ true}\ \oplus R_2$$

$$\text{no } \mathbf{0}R \text{ rule} \qquad \frac{}{\Delta, \mathbf{0}\ res \Vdash C\ true}\ \mathbf{0}L$$

Exponential Connective

$$\frac{\Gamma; \cdot \Vdash A\ true}{\Gamma; \cdot \Vdash\ !A\ true}\ !R \qquad \frac{(\Gamma, A\ ures); \Delta \Vdash C\ true}{\Gamma; \Delta, !A\ res \Vdash C\ true}\ !L$$

Figure 1: Intuitionistic Linear Logic

15-819K: Logic Programming

Lecture 13

Abstract Logic Programming

Frank Pfenning

October 10, 2006

In this lecture we discuss general criteria to judge whether a logic or a fragment of it can be considered a logic programming language. Taking such criteria as absolute is counterproductive, but they might nonetheless provide some useful insight in the design of richer languages. Three criteria emerge: the first two characterize the relationship to logic in that the language should be sound and (non-deterministically) complete, allowing us to interpret both success and finite failure. The third is operational: we would like to be able to interpret connectives in goals as search instructions, giving them a predictable operational semantics.

13.1 Logic and Logic Programming

For a language to claim to be a logic programming language, the first criterion seems to be soundness with respect to the logical interpretation of the program. I consider this non-negotiable: when the interpreter claims something is true, it should be true. Otherwise, it may be a programming language, but the connection to logic has been lost. Prolog, unfortunately, is unsound in this respect, due to the lack of occurs-check and the incorrect treatment of disequality. We either have to hope or verify that these features of Prolog did not interfere with the correctness of the answer. Other non-logical features such as meta-call, cut, or input and output are borderline with respect to this criterion: since these do not have a logical interpretation, it is difficult to assess soundness of such programs, except by reference to an operational semantics.

The second criterion is non-deterministic completeness. This means that if search fails finitely, no proof can exist. This does not seem quite

as fundamental, since we should be mostly interested in obtaining proofs when they exist, but from an abstract perspective it is certainly desirable. Again, this fails for full Prolog, but is satisfied by pure Prolog even with a depth-first interpreter.

Summary: if we would like to abstractly classify logics or logical fragments as suitable basis for logic programming languages, we would expect at least soundness and non-deterministic completeness so we can correctly interpret success and failure of goals.

13.2 Logic Programming as Goal-Directed Search

Soundness and completeness (in some form) establish a connection to logic, but by themselves they are clearly insufficient from a programming perspective. For example, a general purpose theorem prover for a logic is sound and complete, and yet not by itself useful for programming. We would like to ensure, for example, that our implementation of quicksort really is an implementation of quicksort, and similarly for mergesort. The programmer should be able to predict and control operational behavior well enough to cast algorithms into correct implementations.

As we have seen in the case of Prolog, if the language of goals is sufficiently rich, we can transform all the clauses defining a predicate into the form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G$ through a process of residuation. Searching for a proof of a goal $p(\mathbf{t})$ then becomes a procedure call, solving instead $G(\mathbf{t}/\mathbf{x})$, which is another goal. In that way, all computational mechanisms are concentrated on the interpretation of goals. Logic programming, as conceived so far, is goal-directed search. Elevating these observations to a design principle we postulate:

> *An* abstract logic programming language *is defined by a subset of the propositions in a logic together with an operational semantics via proof search on that subset. The operational semantics should be sound, non-deterministically complete, and goal-directed.*

But what exactly does goal-directed search mean? If we consider a sequent $\ldots \Vdash G\ \mathit{true}$ where "$\ldots$" collects all hypotheses (linear, unrestricted, and whatever judgments may arise in other logics), then search is *goal-directed* if we can always break down the structure of the goal G first before considering the hypotheses, including the program. This leads us to the definition of asynchronous connectives.

13.3 Asynchronous Connectives

A logial constant or connective is *asynchronous* if its right rule can always be applied eagerly without loosing completeness. For example, $A \,\&\, B$ is asynchronous, because the rule

$$\frac{\Delta \Vdash A \ true \quad \Delta \Vdash B \ true}{\Delta \Vdash A \,\&\, B \ true}\ \&R$$

can always be used for a conjunctive goal $A \,\&\, B$, rather than first applying a left rule to an assumption Δ or the using a clause in the program Γ. This intuitively coincides with our reading of conjunction as a search instructions: to search for a proof of $A \,\&\, B$, first find a proof of A and then of B. This does not talk about possibly having to decompose the program. The property of being asynchronous turns out to be relatively easy to prove. For example, given $\Delta \Vdash A \,\&\, B \ true$, we can prove that $\Delta \Vdash A \ true$ and $\Delta \Vdash B \ true$ by induction on the structure of the given derivation.

On the other hand, the disjunction $A \oplus B$ (which corresponds to $A \vee B$ in ordinary logic programming) is not asynchronous. As a countexample, consider

$$C, C \multimap (B \oplus A) \Vdash A \oplus B.$$

We can use neither the $\oplus R_1$ nor the $\oplus R_2$ rule, because neither

$$C, C \multimap (B \oplus A) \Vdash A$$

nor

$$C, C \multimap (B \oplus A) \Vdash B$$

are provable. Instead we can prove it as follows in the sequent calculus:

$$\dfrac{\dfrac{}{C \Vdash C}\ \text{id} \quad \dfrac{\dfrac{\dfrac{}{B \Vdash B}\ \text{id}}{B \Vdash A \oplus B}\ \oplus R_2 \quad \dfrac{\dfrac{}{A \Vdash A}\ \text{id}}{A \Vdash A \oplus B}\ \oplus R_1}{B \oplus A \Vdash A \oplus B}\ \oplus L}{C, C \multimap (B \oplus A) \Vdash A \oplus B}\ \multimap L.$$

Observe that we took two steps on the left ($\multimap L$ and $\oplus L$) before decomposing the right.

This counterexample shows that we could not decompose $A \oplus B$ eagerly in goal-directed search, unless we are willing to sacrifice completeness. But what, then, would the program $C, C \multimap (B \oplus A)$ mean?

We have already seen that disjunction is useful in Prolog programs, and the same is true for linear logic programs, so this would seem unfortunate. Before we rescue disjunction, let us analyze which connectives are asynchronous.

We postpone the proofs that the asynchronous connectives are indeed asynchronous, and just give counterexamples for those that are not asynchronous.

$$\begin{array}{lcl} C, C \multimap (B \otimes A) & \Vdash & A \otimes B \\ C, C \multimap \mathbf{1} & \Vdash & \mathbf{1} \\ C, C \multimap (B \oplus A) & \Vdash & A \oplus B \\ C, C \multimap \mathbf{0} & \Vdash & \mathbf{0} \\ C, C \multimap\, !A & \Vdash & !A \end{array}$$

In each case we have to apply two left rules first, before any right rule can be applied.

This leaves $A \multimap B$, $A \,\&\, B$, and $\top$ as asynchronous. Atomic propositions have a somewhat special status in that we cannot decompose them, but we have to switch to the left-hand side and focus on an assumption (which models procedure call).

We have not discussed the rules for linear quantifiers (see below) but it turns out that $\forall x.\, A$ is asynchronous, while $\exists x.\, A$ is not asynchronous.

13.4 Asynchronous Connectives vs. Invertibility of Rules

We call an inference rules *invertible* if the premisses are provable whenever the conclusion is. It is tempting to think that a connective is asynchronous if and only if its right rule is invertible. Not so. Please consider the question and see a counterexample at the end of the lecture notes only in desparation.

13.5 Unrestricted Implication

The analysis so far would suggest that the fragment of our logic has the form

$$A \quad ::= \quad P \mid A_1 \multimap A_2 \mid A_1 \,\&\, A_2 \mid \top \mid \forall x.\, A$$

However, this is insufficient, because it is purely linear. Usually we integrate non-linear reasoning into linear logic using the "*of course*" operator '!', but this is not asynchronous. Instead we can use ordinary implication to complete the picture. The proposition $A \supset B$ is true if assuming A as an

unrestricted resource we can prove B.

$$\frac{\Gamma, A\ \mathit{ures}; \Delta \Vdash B\ \mathit{true}}{\Gamma; \Delta \Vdash A \supset B\ \mathit{true}}\ \supset R \qquad \frac{\Gamma; \cdot \Vdash A\ \mathit{true} \quad \Gamma; \Delta \Vdash C\ \mathit{true}}{\Gamma; \Delta, A \supset B\ \mathit{res} \Vdash C\ \mathit{true}}\ \supset L$$

In the left rule for implication, we cannot use any linear resources to prove A, because A may be used in an unrestricted way when proving B. We would need those resources potentially many times in the proof of B, violating linearity of the overall system.

The implication $A \supset B$ is equivalent to $(!A) \multimap B$ (see Exercise 13.1), but they have different proof theoretic properties since $!A$ is not asynchronous.

13.6 Focusing and Synchronous Connectives

If all goal connectives are asynchronous, then we will hit eventually hit an atomic predicate without even looking at the program. What happens then? Recall from Lecture 8 that we now *focus* on a particular program clause and decompose this in a focusing phase. In the setting here this just means that the left rules are applied in sequence to the proposition in focus until an atomic formula is reached, and this formula then must match the conclusion.

In general, we call a connective *synchronous* if we can continue to focus on its components when it is in focus without losing completeness. Note that in logic programming we focus on assumptions (that is, program clauses or part of the state), so the status of a connective as synchronous or asynchronous has be considered separately depending on whether it occurs as a goal (on the right-hand side) or as an assumption (on the left-hand side). A remarkable fact is that all the connectives that were asynchronous as goals are synchronous as assumptions.

We now write $\Delta; A \ll P$ for a focus on A where we just wrote $\Delta; A \vdash P$ earlier, to avoid potential confusion with other hypothetical judgment forms.

Unlike the decomposition of asynchronous connectives (which is completely mechanical), the decomposition of propositions in focus in the synchronous phase of search involves choices. For example the pair of rules

$$\frac{\Delta; A_1\ \mathit{res} \ll P\ \mathit{true}}{\Delta; A_1 \wedge A_2\ \mathit{res} \ll P\ \mathit{true}}\ \&L_1 \qquad \frac{\Delta; A_2\ \mathit{res} \ll P\ \mathit{true}}{\Delta; A_1 \wedge A_2\ \mathit{res} \ll P\ \mathit{true}}\ \&L_2$$

requires a choice between A_1 and A_2 in the focusing phase of search, and similarly for other connectives.

To restate the focusing property again: it allows us to continue to make choices on the proposition in focus, without reconsidering other assumptions, until we reach an atomic proposition. If that matches the conclusion that branch of the proof succeeds, otherwise it fails.

In the next lecture we will get a hint on how to prove this, although we will not do this in full detail.

We close this section by giving the focusing rules for the asynchronous fragment of linear logic, which is at the heart of our logic programming language. The rules can be found in Figure 1. Unfortunately our motivating example from the earlier lecture does not fall into this fragment. For example, we used both simultaneous conjunction $A \otimes B$ and disjunction $A \oplus B$ as goals which are so far prohibit. We will resurrect them via residuation, not because they truly add expressive power, but they are convenient both for program expression and for compilation.

13.7 Historical Notes

The notion that a logic programming language should be characterized as a fragment of logic with complete goal-directed search originated with Miller, Nadathur, and Scedrov [11] who explored logic programming based on higher-order logic. A revised and expanded version appeared a couple of years later [10]. These papers introduced the term *uniform proofs* for those proofs that work asynchronously on the goal until it is atomic.

Some time later this was generalized by Andreoli and Pareschi who first recognized the potential of linear logic for logic programming [3]. They used a fragment of *classical* linear logic (rather than the intuitionistic linear logic we use here), which does not have a distinguished notion of goal. The language LO was therefore more suited to concurrent and object-oriented programming [1, 5, 4].

Andreoli also generalized the earlier notion of uniform proofs to *focusing proofs* [2], capturing now both the asynchronous as well as the synchronous behavior of connectives in a proof-theoretic manner. This seminal work subsequently had many important applications in logic, concurrency, and functional programming, and not just in logic programming.

The thread of research on intuitionistic, goal-directed logic programming resumed with the work by Hodas and Miller [8, 9] who proposed essentially what we presented in this lecture, with some additional extralogical features borrowed from Prolog. In honor of its central new connective the language was called Lolli. These ideas were later picked up in the design of a linear logical framework [6, 7] which augments Lolli with a

richer quantification and explicit proof terms.

13.8 Exercises

Exercise 13.1 *Prove that $A \supset B$ is equivalent to $(!A) \multimap B$ in the sense that each entails the other, that is, $A \supset B \Vdash (!A) \multimap B$ and vice versa.*

Exercise 13.2 *At one point we defined pure Prolog with the connectives $A \wedge B$, $\top$, $A \vee B$, $\bot$, $A \supset B$, and $\forall x.\, A$, suitably restricted into legal goals and programs. Show how to translate such programs and goals into linear logic so that focusing proofs for (non-linear) logic are mapped isomorphically to focusing proofs in linear logic, and prove that your translation is correct in that sense.*

13.9 Answer

Consider the right rule for $!A$.

$$\frac{\Gamma; \cdot \Vdash A\ true}{\Gamma; \cdot \Vdash !A\ true}\ !R$$

This rule is invertible: whenever the conclusion is provable, then so is the premiss. However, $!A$ is not asynchronous (see the counterexample in this lecture). If we had formulated the rule as

$$\frac{\Delta = (\cdot) \quad \Gamma; \cdot \Vdash A\ true}{\Gamma; \Delta \Vdash !A\ true}\ !R$$

we would have recognized it as not being invertible, because Δ is not necessarily empty.

13.10 References

[1] Jean-Marc Andreoli. *Proposal for a Synthesis of Logic and Object-Oriented Programming Paradigms*. PhD thesis, University of Paris VI, 1990.

[2] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] Jean-Marc Andreoli and Remo Pareschi. LO and behold! Concurrent structured processes. In *Proceedings of OOPSLA'90*, pages 44–56, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices, vol.25, no.10.

[4] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[5] Jean-Marc Andreoli and Remo Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[6] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002.

[8] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 32–42, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[9] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[10] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[11] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

Judgmental Rules

$$\frac{}{P\ res \ll P\ true}\ \text{id}$$

$$\frac{A\ ures \in \Gamma \quad \Gamma; \Delta; A\ res \ll P\ true}{\Gamma; \Delta \Vdash P\ true}\ \text{copy} \qquad \frac{\Delta; A\ res \ll P\ true}{\Delta, A\ res \Vdash P\ true}\ \text{focus}$$

Multiplicative Connective

$$\frac{\Delta, A\ res \Vdash B\ true}{\Delta \Vdash A \multimap B\ true} \multimap R \qquad \frac{\Delta_A \Vdash A\ true \quad \Delta_B; B\ res \ll P\ true}{\Delta_A, \Delta_B; A \multimap B\ res \ll P\ true} \multimap L$$

Additive Connectives

$$\frac{\Delta \Vdash A\ true \quad \Delta \Vdash B\ true}{\Delta \Vdash A \,\&\, B\ true}\ \&R \qquad \frac{\Delta; A\ res \ll P\ true}{\Delta; A \,\&\, B\ res \ll P\ true}\ \&L_1 \qquad \frac{\Delta; B\ res \ll P\ true}{\Delta; A \,\&\, B\ res \ll P\ true}\ \&L_2$$

$$\frac{}{\Delta \Vdash \top\ true}\ \top R \qquad \text{no } \top L \text{ rule}$$

Exponential Connective

$$\frac{(\Gamma, A\ ures); \Delta \Vdash B\ true}{\Gamma; \Delta \Vdash A \supset B\ true} \supset R \qquad \frac{\Gamma; \cdot \Vdash A\ true \quad \Gamma; B\ res \ll P\ true}{\Gamma; \Delta; A \supset B\ res \ll P\ true} \supset L$$

Quantifier

$$\frac{\Delta \Vdash A \quad x \notin \mathrm{FV}(\Gamma; \Delta)}{\Delta \Vdash \forall x.\, A}\ \forall R \qquad \frac{\Delta; A(t/x)\ res \ll P\ true}{\Delta; \forall x.\, A\ res \ll P\ true}\ \forall L$$

Figure 1: Focused Intuitionistic Linear Logic; Asynchronous Fragment

15-819K: Logic Programming

Lecture 14

Cut Elimination

Frank Pfenning

October 12, 2006

In this lecture we consider how to prove that connectives are asynchronous as goals and then consider cut elimination, one of the most important and fundamental properties of logical systems. We then revisit residuation to restore some of the connectives not present in the asynchronous fragment of linear logic. For each synchronous assumption we find a corresponding synchronous goal connective.

14.1 Proving Connectives Asynchronous

We have claimed in the last lecture that certain connectives of linear logic are asynchronous as goals in order to justify their inclusion in our linear logic programming language. I know of essentially two ways to prove that such operators are indeed asynchronous. The first is by simple inductions, one for each asynchronous connectives. The following theorem provides an example. In todays lecture we generally omit the judgment form for propositions such as true on the right-hand side, and res or ures on the left-hand side, since this can be infered from the placement of the proposition.

Theorem 14.1 *If $\Delta \Vdash A \mathbin{\&} B$ then $\Delta \Vdash A$ and $\Delta \Vdash B$.*

Proof: By induction on the structure of $\mathcal{D}$, the deduction of $\Delta \Vdash A \mathbin{\&} B$. We show only two cases; others are similar.

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Delta \Vdash A \quad \Delta \Vdash B} \mathbin{\&}R.$

$\Delta \Vdash A$	Subderivation $\mathcal{D}_1$
$\Delta \Vdash B$	Subderivation $\mathcal{D}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Delta', C_1 \Vdash A \mathbin{\&} B\end{array}}{\Delta', C_1 \mathbin{\&} C_2 \Vdash A \mathbin{\&} B}\ \&L_1$ where $\Delta = (\Delta', C_1 \mathbin{\&} C_2)$.

$\Delta', C_1 \Vdash A$ and	
$\Delta', C_1 \Vdash B$	By i.h. on $\mathcal{D}_1$
$\Delta', C_1 \mathbin{\&} C_2 \Vdash A$	By rule $\&L_1$
$\Delta', C_1 \mathbin{\&} C_2 \Vdash B$	By rule $\&L_1$

□

There is a second way to proceed, using the admissibility of cut from the next section directly, without appeal to induction.

14.2 Admissibility of Cut

So far we have not justified that the right and left rules for the connectives actually match up in an expected way. What we would like to show is that the judgment of a being resource and the judgment of truth, when combined in a linear hypothetical judgment, coincide. There are two parts to this. First, we show that with the resource A we can achieve the goal A, for arbitrary A (not just atomic predicates).

Theorem 14.2 (Identity Principle) *A res $\Vdash$ A true for any proposition A.*

Proof: See Exercise 12.1. □

Second, we show that if we can achieve A as a goal, it is legitimate to assume A as a resource. This completes the theorems which show our sequent calculus is properly designed.

Theorem 14.3 (Admissibility of Cut)
If $\Delta_A \Vdash A$ true and Δ_C, A res $\Vdash C$ true then $\Delta_C, \Delta_A \Vdash C$ true.

Proof: We prove this here only for the purely linear fragment, without the operators involving unrestricted resources (!A, $A \supset B$). The proof proceeds by nested induction, first on the structure of A, the so-called *cut formula*, then simultaneously on the structure of $\mathcal{D}$, the derivation of $\Delta_A \Vdash A$ *true* and $\mathcal{E}$, the derivation of Δ_C, A *res* $\Vdash C$ *true*. This form of induction means we can appeal to the induction hypothesis

1. either on a smaller cut formula with arbitrary derivations, or

2. on the same cut formula A and same $\mathcal{D}$, but a subderivation of $\mathcal{E}$, or

3. on the same cut formula A and same $\mathcal{E}$, but a subderivation of $\mathcal{D}$.

There are many cases to distinguish; we show only three which illustrate the reasons behind the form of the induction above.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta_A \Vdash A_1 & \Delta_A \Vdash A_2\end{array}}{\Delta_A \Vdash A_1 \mathbin{\&} A_2}\ \&R$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Delta_C, A_1 \Vdash C\end{array}}{\Delta_C, A_1 \mathbin{\&} A_2 \Vdash C}\ \&L_1$.

$\Delta_C, \Delta_A \Vdash C$ By i.h. on A_1, $\mathcal{D}_1$ and $\mathcal{E}_1$

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta_1 \Vdash A_1 & \Delta_2 \Vdash A_2\end{array}}{\Delta_1, \Delta_2 \Vdash A_1 \otimes A_2}\ \otimes R$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}' \\ \Delta_C, A_1, A_2 \Vdash C\end{array}}{\Delta_C, A_1 \otimes A_2 \Vdash C}\ \otimes L$.

$\Delta_C, A_1, \Delta_2 \Vdash C$ By i.h. on A_2, $\mathcal{D}_2$, and $\mathcal{E}'$
$\Delta_C, \Delta_1, \Delta_2 \Vdash C$ By i.h. on A_1, $\mathcal{D}_1$ and the previous line
$\Delta_C, \Delta_A \Vdash C$ Since $\Delta_A = (\Delta_1, \Delta_2)$

Case: $\begin{array}{c}\mathcal{D} \\ \Delta_A \Vdash A\end{array}$ is arbitrary and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Delta', D_1, A \Vdash C\end{array}}{\Delta', D_1 \mathbin{\&} D_2, A \Vdash C}\ \&L_1$.

$\Delta', D_1, \Delta_A \Vdash C$ By i.h. on A, $\mathcal{D}$ and $\mathcal{E}_1$
$\Delta', D_1 \mathbin{\&} D_2, \Delta_A \Vdash C$ By rule $\&L_1$

□

The shown cases are typical in that if the cut formulas were just introduced on both the right and the left, then we can appeal to the induction hypothesis on its subformulas. Otherwise, we can keep the cut formula and either $\mathcal{D}$ or $\mathcal{E}$ the same and appeal to the induction hypothesis on the subderivations on the other side.

The case for $\otimes$ above shows why we cannot simply use an induction over the derivations $\mathcal{D}$ and $\mathcal{E}$, because the second time we appeal to the induction hypothesis, one of the derivations come from a previous appeal to the induction hypothesis and could be much larger than $\mathcal{E}$.

The proof is not too difficult to extend to the case with unrestricted assumptions (see Exercise 14.2).

14.3 Cut Elimination

Cut elimination is the property that if we take cut as a new inference rule, it can be eliminated from any proof. Actually, here we would have two cut rules.

$$\frac{\Gamma; \Delta_A \Vdash A\ true \quad \Gamma; \Delta_C, A\ res \Vdash C\ true}{\Gamma; \Delta_C, \Delta_A \Vdash C\ true}\ \text{cut}$$

$$\frac{\Gamma; \cdot \Vdash A\ true \quad (\Gamma, A\ ures); \Delta \Vdash C\ true}{\Gamma; \Delta \Vdash C\ true}\ \text{cut!}$$

Showing that cut can be eliminated is an entirely straightforward induction over the structure of the deduction with cuts. In each case we just appeal to the induction hypothesis on each premiss and re-apply the rule to the get the same conclusion. The only exception are the cut rules, in which case we obtain cut-free derivations of the premisses by induction hypothesis and then appeal to the admissibility of cut to get a cut-free derivation of the conclusion.

Cut as a new rule, however, is unfortunate from the perspective of proof search. When read bottom-up, we have to invent a new proposition A, which we then prove. When this proof succeeds we would be allowed to assume it into our overall proof. While mathematically inventing such lemmas A is critical, in a logic programming language it destroys the goal-directed character of search.

14.4 Asynchronous Connectives, Revisited

Using cut elimination we can give alternate proofs that connectives are asynchronous. We show only one example, for conjunction.

Theorem 14.4 *If $\Delta \Vdash A \,\&\, B$ then $\Delta \Vdash A$ and $\Delta \Vdash B$.*

Proof: (Alternate) Direct, using admissibility of cut.

$\Delta \Vdash A \,\&\, B$	Given
$A \Vdash A$	Identity principle
$A \,\&\, B \Vdash A$	By rule $\&L_1$
$\Delta \Vdash A$	By admissibility of cut
$B \Vdash B$	Identify principle
$A \,\&\, B \Vdash B$	By rule $\&L_2$
$\Delta \Vdash B$	By admissibility of cut

□

14.5 Residuation and Synchronous Goal Connectives

In the focusing calculus from the previous lecture, all connectives are asynchronous as goals and synchronous when in focus as assumptions. In our little programming example of peg solitaire, we extensively used simultaneous conjunction ($\otimes$) and and also disjunction ($\oplus$). One question is how to extend our language to include these connectives. So far, we have, for both programs and goals:

$$A \quad ::= \quad P \mid A_1 \multimap A_2 \mid A_1 \mathbin{\&} A_2 \mid \top \mid \forall x.\, A \mid A_1 \supset A_2$$

A principled way to approach this question is to return to residuation. Given a program clause this constructs a goal whose search behavior is equivalent to the behavior of the clause. Since we have already seen residuation in detail for the non-linear case, we just present the rules here.

$$\frac{}{P' \Vdash P > P' \doteq P} \qquad \frac{D_1 \Vdash P > G_1}{G_2 \multimap D_1 \Vdash P > G_1 \otimes G_2}$$

$$\frac{D_1 \Vdash P > G_1 \quad D_2 \Vdash P > G_2}{D_1 \mathbin{\&} D_2 \Vdash P > G_1 \oplus G_2} \qquad \frac{}{\top \Vdash P > \mathbf{0}}$$

$$\frac{D \Vdash P > G \quad x \notin \mathrm{FV}(P)}{\forall x.\, D \Vdash P > \exists x.\, G} \qquad \frac{D_1 \Vdash P > G_1}{G_2 \supset D_1 \Vdash P > G_1 \otimes\!!\, G_2}$$

There are a few connectives we have not seen in their full generality in linear logic, namely equality, existential quantification, and a curious asymmetric connective $G_1 \otimes\!!\, G_2$. We concentrate here on their behavior as goals (see Exercise 14.5). Because these connectives mirror the synchronous behavior of the assumption in focus, proving one of these is now a focusing judgment, except that we focus on a goal. We write this as $\Gamma; \Delta \gg G$.

First, in our proof search judgment we replace the focus and copy rules by appeals to residuation.

$$\frac{D \in \Gamma \quad D \Vdash P > G \quad \Gamma; \Delta \gg G}{\Gamma; \Delta \Vdash P} \text{ resid!}$$

$$\frac{D \Vdash P > G \quad \Gamma; \Delta \gg G}{\Gamma; \Delta, D \Vdash P} \text{ resid}$$

Next the rules for focusing on the right.

$$\frac{\Delta = (\cdot)}{\Delta \gg P \doteq P}\ \text{id} \qquad \frac{\Delta = (\Delta_1, \Delta_2) \quad \Delta_1 \gg G_1 \quad \Delta_2 \gg G_2}{\Delta \gg G_1 \otimes G_2}\ \otimes R$$

$$\frac{\Delta \gg G_1}{\Delta \gg G_1 \oplus G_2}\ \oplus R_1 \qquad \frac{\Delta \gg G_2}{\Delta \gg G_1 \oplus G_2}\ \oplus R_2 \qquad \text{no } \mathbf{0}R \text{ rule for } \Delta \gg \mathbf{0}$$

$$\frac{\Delta \gg G(t/x)}{\Delta \gg \exists x.\, G}\ \exists R \qquad \frac{\Gamma; \Delta \gg G_1 \quad \Gamma; \cdot \gg G_2}{\Gamma; \Delta \gg G_1 \otimes! \, G_2}\ \otimes! R$$

Furthermore, we transition back to asynchronous decomposition when we encounter an asynchronous connective. We call this *blurring* the focus. Conversely, we focus on the right when encountering a synchronous connective.

$$\frac{\Delta \Vdash G \quad G \text{ asynch.}}{\Delta \gg G}\ \text{blur} \qquad \frac{\Delta \gg G \quad G \text{ synch.}}{\Delta \Vdash G}\ \text{rfocus}$$

For completeness, we give the remaining rules for asynchronous goals (the atomic case is above in the resid and resid! rules).

$$\frac{\Delta, D_1 \Vdash G_2}{\Delta \Vdash D_1 \multimap G_2}\ \multimap R \qquad \frac{\Delta \Vdash G_1 \quad \Delta \Vdash G_2}{\Delta \Vdash G_1 \,\&\, G_1}\ \& R \qquad \frac{}{\Delta \Vdash \top}\ \top R$$

$$\frac{\Delta \Vdash G \quad x \notin \mathrm{FV}(\Gamma; \Delta)}{\Delta \Vdash \forall x.\, G}\ \forall R \qquad \frac{(\Gamma, D_1); \Delta \Vdash G_2}{\Gamma; \Delta \Vdash D_1 \supset G_2}\ \supset R$$

This yields the following grammar of so-called *linear hereditary Harrop formulas* which form the basis of the Lolli language. The fragment without $\multimap$ and $\otimes$, replacing $\wedge/\&, \vee/\oplus, \bot/\mathbf{0}, \wedge/\otimes!$, is called *hereditary Harrop formulas* and forms the basis for λProlog.

$$\begin{array}{llcl}
\text{Goals} & G & ::= & \\
\quad \text{Asynch.} & & P & \mid D_1 \multimap G_2 \mid G_1 \,\&\, G_2 \mid \top \mid \forall x.\, G \mid D_1 \supset G_2 \\
\quad \text{Synch.} & \mid & P' \doteq P \mid G_1 \otimes G_2 \mid G_1 \oplus G_2 \mid \mathbf{0} \mid \exists x.\, A \mid G_1 \otimes! \, G_2 \\
\text{Programs} & D & ::= & P \mid G_2 \multimap D_1 \mid D_1 \,\&\, D_2 \mid \top \mid \forall x.\, D \mid G_2 \supset D_1
\end{array}$$

We have lined up the synchronous goals with their counterparts as synchronous programs just below, as explained via residuation.

Strictly speaking, going back and forth between the $\Delta \Vdash G$ and $\Delta \gg G$ is unnecessary: we could coalesce the two into one because programs are

always fully synchronous. However, it highlights the difference between the synchronous and asynchronous right rules: Asynchronous decomposition in $\Delta \Vdash G$ is automatic and involves no choice, synchronous decomposition $\Delta \gg G$ involves a significant choice and may fail. Moreover, in just about any logical extension of focusing beyond this fragment, we need to pause when the goal becomes synchronous during the asynchronous decomposition phase and consider whether to focus on an assumption instead. Here, this would always fail.

In practice, it is convenient to admit an even slightly richer set of goals, whose meaning can be explained either via a transformation to the connectives already shown above or directly via synchronous or asynchronous rules for them (see Exercise 14.3).

14.6 Completeness of Focusing

Soundness of the focusing system is easy to see, since each rule is a restriction of the corresponding left and right rules for the non-focused sequent calculus. Completeness is somewhat more difficult. We can continue the path mapped out in the proof that various connectives are asynchronous as goals, proving that the same connectives are indeed synchronous as programs. Alternatively, we can prove a generalization of the cut elimination results for focused derivations and use that in an overall completeness proof. The references below give some pointers to the two different styles of proof in the literature.

14.7 Historical Notes

Cut elimination, one of the most fundamental theorems in logic, is due to Gentzen [3], who introduced the sequent calculus and natural deduction for both classical and intuitionistic logic and showed cut elimination. His formulation of the sequent calculus had explicit rules for exchange, weaking, and contraction, which make the proof somewhat more tedious than the one we presented here. I first provided proofs by simple nested structural induction, formalized in a logical framework, for intuitionistic and classical [5, 6] as well as linear logic [4].

Andreoli introduced focusing for classical linear logic [1] and proved its completeness through a number of inversion and admissibility properties. An alternative proof using cut elimination as a central lemma, applied to intuitionistic linear logic was given by Chaudhuri [2].

14.8 Exercises

Exercise 14.1 *Prove $A \multimap B$ to be asynchronous on the right in two ways:*

i. directly by induction, and

ii. by appealing to the admissibility of cut.

Exercise 14.2 *In order to prove the cut elimination theorem in the presence of unrestricted assumptions, we generalize to the following:*

1. *(Cut) If* $\Gamma; \Delta_A \Vdash A\ true$ *and* $\Gamma; \Delta_C, A\ res \Vdash C\ true$ *then* $\Gamma; \Delta_C, \Delta_A \Vdash C\ true$.
2. *(Cut!) If* $\Gamma; \cdot \Vdash A\ true$ *and* $(\Gamma, A\ ures); \Delta_C \Vdash C\ true$ *then* $\Gamma; \Delta_C \Vdash C\ true$

The second form of cut expresses that if we can prove A without using resources, it is legitimate to assume it as an unrestricted resource, essentially because we can generate as many copies of A as we need (it requires no resources).

The nested induction now proceeds first on the structure of the cut formula A, then on the form of cut where cut $<$ cut!, then simultaneously on the structures of the two given derivations $\mathcal{D}$ and $\mathcal{E}$. This means we can appeal to the induction hypothesis

1. *either on a subformula of A with arbitrary derivations, or*
2. *on the same formula A where cut! appeals to cut, or*
3. *on the same cut formula and same form of cut and same $\mathcal{D}$, but a subderivation of $\mathcal{E}$, or*
4. *on the same cut formula and same form of cut and same $\mathcal{E}$, but a subderivation of $\mathcal{D}$.*

Show the cases explicitly involving $!A$, $A \supset B$, and copy in this proof. You may assume that weakening the unrestricted assumptions by adding more is legitimate and does not change the structure of the given deduction. Note carefully appeals to the induction hypothesis and explain why they are legal.

Exercise 14.3 *We consider even larger set of goals to squeeze the last bit of convenience out of our language without actually affecting its properties.*

i. Give the rule(s) to allow $!G$ as a goal.

ii. Give the rule(s) to allow $\mathbf{1}$ *as a goal.*

iii. We could allow simultaneous conjunction on the left-hand side of linear implication goals, because $(D_1 \otimes D_2) \multimap G$ *is equivalent to* $D_1 \multimap (D_2 \multimap G)$*, which lies within the permitted fragment. Explore which formulas* R *could be allowed in goals of the form* $R \multimap G$ *because they can be eliminated by a local equivalence-preserving transformation such as the one for* $\otimes$*.*

iv. Now explore which formulas S *could be allowed in goals of the form* $S \supset G$ *without affecting the essence of the language.*

Exercise 14.4 *Prove that the focusing system with left and right rules is equivalent to the system with only right rules and residuation for atomic goals.*

Exercise 14.5 *Through residuation, we have introduced two new connectives to linear logic,* $A \otimes\!! B$ *and* $P' \doteq P$*, but we have only considered their right rules.*

Give corresponding left rules for them in the sequent calculus and prove cut elimination and identity for your rules.

14.9 References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, 2006. To appear.

[3] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[4] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[5] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, November 1994.

[6] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

15-819K: Logic Programming

Lecture 15

Resource Management

Frank Pfenning

October 17, 2006

In this lecture we address the resource management problem in linear logic programming. We also give some small but instructive examples of linear logic programming, supplementing the earlier peg solitaire code.

15.1 Input/Output Interpretation of Resources

Reconsider the rule for simultaneous conjunction as a goal.

$$\frac{\Delta = (\Delta_1, \Delta_2) \quad \Delta_1 \Vdash G_1 \quad \Delta_2 \Vdash G_2}{\Delta \Vdash G_1 \otimes G_2} \otimes R$$

The difficulty in using this rule in proof search is that, as written, we have to "guess" the right way to split the resources Δ into two. Clearly, enumerating all possible ways to split Δ will be very inefficient, and also difficult for the programmer to predict.

Instead, we pass all resources to the goal G_1 and then pass the ones that were not consumed in the derivation of G_1 to G_2. We write

$$\Delta_I \setminus \Delta_O \Vdash G$$

where Δ_I is the input context and Δ_O is the output context generated by the proof search for G. The invariant we preserve is that

$$\Delta_I \setminus \Delta_O \Vdash G \text{ iff } \Delta_I - \Delta_O \Vdash G$$

where $\Delta_I - \Delta_O$ subtracts the resources in Δ_O from the resources in Δ_I. It is convenient to keep Δ_I and Δ_O ordered, so that this difference can be computed component by component (see below).

Now the rule for simultaneous conjunction is

$$\frac{\Delta_I \setminus \Delta_M \Vdash G_1 \quad \Delta_M \setminus \Delta_O \Vdash G_2}{\Delta_I \setminus \Delta_O \Vdash G_1 \otimes G_2} \otimes R.$$

It is easy to verify that the above invariant holds for this rule. More formally, this would be part of a soundness and completeness proof for the input/output interpretation of resources.

15.2 Slack Resources

The simple input/output interpretation for resources breaks down for consumptive truth ($\top$). Recall the rule

$$\frac{}{\Delta \Vdash \top} \top R$$

which translates to

$$\frac{\Delta_I \supseteq \Delta_O}{\Delta_I \setminus \Delta_O \Vdash \top} \top R$$

because $\top$ *may* consume any of its input but does *not need* to. Now we are back at the original problem, since we certainly do not want to enumerate all possible subsets of Δ_I blindly.

Instead, we pass on all the input resources, but also a flag to indicate that all of these resources could have been consumed. That means if they are left over at the end, we can succeed instead of having to fail. We write the judgment as

$$\Delta_I \setminus \Delta_O \Vdash_v G$$

where $v = 0$ means G used exactly the resources in $\Delta_I - \Delta_O$, while $v = 1$ means G could also have consumed additional resources from Δ_O. Now our invariants are:

i. $\Delta_I \setminus \Delta_O \Vdash_0 G$ iff $\Delta_I - \Delta_O \Vdash G$

ii. $\Delta_I \setminus \Delta_O \Vdash_1 G$ iff $\Delta_I - \Delta_O, \Delta' \Vdash G$ for any $\Delta_O \supseteq \Delta'$.

The right rule for $\top$ with slack is just

$$\frac{}{\Delta_I \setminus \Delta_I \Vdash_1 \top} \top R.$$

In contrast, the rule for equality which requires the linear context to be empty is

$$\frac{}{\Delta_I \setminus \Delta_I \Vdash_0 t \doteq t} \doteq R.$$

As far as resources are concerned, the only difference is whether slack is allowed ($v = 1$) or not ($v = 0$).

We now briefly return to simultaneous conjunction. There is slack in the deduction for $G_1 \otimes G_2$ if there is slack on either side: any remaining resources could be pushed up into either of the two subderivations as long as there is slack in at least one.

$$\frac{\Delta_I \setminus \Delta_M \Vdash_v G_1 \quad \Delta_M \setminus \Delta_O \Vdash_w G_2}{\Delta_I \setminus \Delta_O \Vdash_{v \vee w} G_1 \otimes G_2} \otimes R.$$

Here $v \vee w$ is the usual Boolean disjunction between the two flags: it is 0 if both disjuncts are 0 and 1 otherwise.

15.3 Strict Resources

Unfortunately, there is still an issue in that resource management does not take into account all information it should. There are examples in the literature [2], but they are not particularly natural. For an informal explanation, consider the overall query $\Delta \Vdash G$. We run this as $\Delta \setminus \Delta_O \Vdash_v G$, where Δ_O and v are returned. We then have to check that all input has indeed be consumed by verifying that Δ_O is empty. If Δ_O is not empty, we have to fail this attempt and backtrack.

We would like to achieve that we fail as soon as possible when no proof can exist due to resource management issues. In the present system we may sometimes run to completion only to note at that point that we failed to consume all resources. We can avoid this issue by introducing yet one more distinction into our resource management judgment by separating out *strict resources*. Unlike Δ_I, which represents resources which *may* be used, Ξ represent resources which *must* be used in a proof.

$$\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$$

The invariant does not get significantly more complicated.

i. $\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G$ iff $\Xi, \Delta_I - \Delta_O \Vdash G$

ii. $\Xi; \Delta_I \setminus \Delta_O \Vdash_1 G$ iff $\Xi, \Delta_I - \Delta_O, \Delta' \Vdash G$ for all $\Delta_O \supseteq \Delta'$.

When reading the rules please remember that no resource in Ξ is ever passed on: it *must* be consumed in the proof of $\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$. The unrestricted context Γ remains implicit and is always passed from conclusion to all premisses.

We will enforce as an additional invariant that input and output context have the same length and structure, except that some inputs have been consumed. Such consumed resources are noted as underscores '_' in a context. We use $\Delta_I \supseteq \Delta_O$ and $\Delta_I - \Delta_O$ with the following definitions:

$$\frac{}{(\cdot) \supseteq (\cdot)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, \_) \supseteq (\Delta_O, \_)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, A) \supseteq (\Delta_O, \_)} \qquad \frac{\Delta_I \supseteq \Delta_O}{(\Delta_I, A) \supseteq (\Delta_O, A)}$$

and, for $\Delta_I \supseteq \Delta_O$,

$$\frac{}{(\cdot) - (\cdot) = (\cdot)} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, \_) - (\Delta_O, \_) = (\Delta, \_)}$$

$$\frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, A) - (\Delta_O, \_) = (\Delta, A)} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, A) - (\Delta_O, A) = (\Delta, \_)}$$

Atomic Goals. When a goal is atomic, we focus on an assumption, residuate, and solve the resulting subgoal. There are three possibilities for using an assumption: from Γ, from Ξ, or from Δ_I. Because we use residuation, resource management is straightforward here: we just have to replace the assumption with the token '_' to indicate that the resource has been consumed.

$$\frac{D \in \Gamma \quad D \Vdash P > G \quad \Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v P} \text{ resid!}$$

$$\frac{D \Vdash P > G \quad \Gamma; (\Xi_1, \_, \Xi_2); \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; (\Xi_1, D, \Xi_2); \Delta_I \setminus \Delta_O \Vdash_v P} \text{ resid}_1$$

$$\frac{D \Vdash P > G \quad \Gamma; \Xi; (\Delta_I', \_, \Delta_I'') \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; (\Delta_I', D, \Delta_I'') \setminus \Delta_O \Vdash_v P} \text{ resid}_2$$

Asynchronous Multiplicative Connective. There is only one multiplicative asynchronous connective, $D \multimap G$ which introduces a new linear assumption. Since D *must* be consumed in the proof of G, we add it to the

strict context Ξ.

$$\frac{(\Xi, D); \Delta_I \setminus \Delta_O \Vdash_v G}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v D \multimap G} \multimap R$$

Synchronous Multiplicative Connectives. In the linear hereditary Harrop fragment as we have constructed it here, there are only two multiplicative connectives that are synchronous as goals: equality and simultaneous conjunction. The multiplicative unit **1** is equivalent to $P \doteq P$ and does not explicitly arise in residuation. For equality, we just need to check that Ξ is empty and pass on all input resources to the output, indicating that there is no slack ($v = 0$).

$$\frac{\Xi = (\_, \ldots, \_)}{\Xi; \Delta_I \setminus \Delta_I \Vdash_0 P \doteq P} \doteq R$$

For simultaneous conjunction, we distinguish two cases, depending on whether the first subgoal has slack. Either way, we turn all strict resources from Ξ into lax resources for the first subgoal, since the second subgoal is waiting, and may potentially consume some of the formulas in Ξ that remain unconsumed in the first subgoal.

$$\frac{.\,; \Xi, \Delta_I \setminus \Xi', \Delta'_I \Vdash_0 G_1 \quad \Xi'; \Delta'_I \setminus \Delta_O \Vdash_v G_2 \quad (\Xi \supseteq \Xi')}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \otimes G_2} \otimes R_0$$

If the first subgoal is slack, then it could consume the leftover resources in Ξ', so they do not necessarily need to be consumed in the second subgoal. Originally strict resources that remain after the second subgoal are then dropped, noting that the first subgoal must have (implicitly) consumed them.

$$\frac{.\,; \Xi, \Delta_I \setminus \Xi', \Delta'_I \Vdash_1 G_1 \quad .\,; \Xi', \Delta'_I \setminus \Xi'', \Delta_O \Vdash_* G_2 \quad (\Xi \supseteq \Xi' \supseteq \Xi'')}{\Xi; \Delta_I \setminus \Delta_O \Vdash_1 G_1 \otimes G_2} \otimes R_1$$

It does not matter whether the second subgoal is strict or lax, since the disjunction is already known to be 1. We indicate this with an asterisk '$*$'.

Asynchronous Additive Connectives. The additive connectives that are asynchronous as goals are alternative conjunction ($G_1 \,\&\, G_2$) and consumptive truth ($\top$). First $\top$, which motivated the slack indicator v. It consumes all of Ξ and passes the remaining inputs on without consuming them.

$$\frac{}{\Xi; \Delta_I \setminus \Delta_I \Vdash_1 \top} \top R$$

For alternative conjunction, we distinguish two subcases, depending on whether the first subgoal turns out to have slack or not. If not, then the second subgoal must consume exactly what the first subgoal consumed, namely Ξ and $\Delta_I - \Delta_O$. We therefore add this to the strict context. The lax context is empty, and we do not need to check the output (it must also be empty, since it is a subcontext of the empty context). Again, we indicate this with a '$*$' to denote an output we ignore.

$$\frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G_1 \quad \Xi, \Delta_I - \Delta_O; \cdot \setminus * \Vdash_* G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_0 G_1 \,\&\, G_2}\ \&R_0$$

If the first subgoal has slack, with still must consume everything that was consumed in the first subgoal. In addition, we may consume anything that was left.

$$\frac{\Xi; \Delta_I \setminus \Delta_M \Vdash_1 G_1 \quad \Xi, \Delta_I - \Delta_M; \Delta_M \setminus \Delta_O \Vdash_v G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \,\&\, G_2}\ \&R_1$$

Synchronous Additive Connectives. Disjunction is easy, because it involves a choice among alternatives, but not resources, which are just passed on.

$$\frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \oplus G_2}\ \oplus R_1 \qquad \frac{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_2}{\Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \oplus G_2}\ \oplus R_2$$

Falsehood is even easier, because it represents failure and therefore has no right rule.

$$\frac{\text{no rule } \mathbf{0}R}{\Xi; \Delta_I \setminus * \Vdash_* \mathbf{0}}$$

Exponential Connectives. Unrestricted implication is quite simple, since we just add the new assumption to the unrestricted context.

$$\frac{\Gamma, D; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v D \supset G}\ \supset R$$

The (asymmetric) exponential conjunction passes all resources to the first subgoal, since the second cannot use any resources. We do not care if the exponential subgoal is strict or lax, since it does not receive or return any resources anyway.

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \quad \Gamma; \cdot; \cdot \setminus * \Vdash_* G_2}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G_1 \otimes! G_2}\ \otimes! R$$

Quantifiers. Quantifiers are resource neutral.

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G \quad x \notin \mathrm{FV}(\Gamma; \Xi; \Delta_I)}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v \forall x.\, G}\ \forall R$$

$$\frac{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v G(t/x)}{\Gamma; \Xi; \Delta_I \setminus \Delta_O \Vdash_v \exists x.\, G}\ \exists R$$

This completes the connectives for the linear hereditary Harrop formulas. The proof that these are sound and complete amounts to showing the invariants stated at the beginning of this section. A crucial lemma states that resources can be added to the lax input context without affecting provability.

If $\Xi; \Delta_I \setminus \Delta_O \Vdash_v G$ *then* $\Xi; (\Delta_I, \Delta') \setminus (\Delta_O, \Delta') \Vdash_v G$ *for all* Δ'.

This provides a measure of modularity to proofs using consumable resources, at least as far as the existence of proofs is concerned. During proof search, however, it is clear that Δ' could interfere with with the proof if added to the input.

At the top level, we solve $\Delta \Vdash G$ by invoking $\Delta; \cdot \setminus * \Vdash_* G$. We do not need to check the output context (which will be empty) or the slack indicator, because Δ is passed in as strict context.

15.4 Sample Program: Permutation

To illustrate linear logic programming we give a few small programs. The first computes all permutations of a list. It does so by adding the elements to the linear context and then reading them out. Since the linear context is not ordered, this allows all permutations.

```
perm([X|Xs],Ys) ⟜ (elem(X) ⊸ perm(Xs,Ys)).
perm([],[Y|Ys]) ⟜ elem(Y) ⊗ perm([],Ys).
perm([],[]).
```

The last clause can only apply if the context is empty, so any order of these clauses will work. However, putting the third before the second will cause more backtracking especially if permutation is embedded multiplicatively in a larger program.

15.5 Sample Program: Depth-First Search

Imperative algorithms for depth-first search mark nodes that have been visited to prevent looping. We can model this in a linear logic program by starting with a linear assumption `node(`x`)` for every node x and consuming this assumption when visiting a node. This means that a node cannot be used more than once, preventing looping.

We assume a predicate `edge(`x`,`y`)` which holds whenever there is a directed edge from x to y.

```
dfs(X,Y) ⟜ edge(X,Y).
dfs(X,Y) ⟜ edge(X,Z) ⊗ node(Z) ⊗ dfs(Z,Y).
```

This by itself is note quite enough because not all nodes might be visited. We can allow this with the following top-level call

```
path(X,Y) ⟜ node(X) ⊗ dfs(X,Y) ⊗ ⊤.
```

15.6 Sample Program: Stateful Queues

In Lecture 11 we have seen how to implement a queue with a difference list, but we had to pass the queue around as an argument to any predicate wanting to use it. We can also maintain the queue in the linear context. Recall that we used a list of instructions `enq(`x`)` and `deq(`x`)`, and that at the end the queue must be empty.

```
queue(Is) ⟜ (front(B) ⊗ back(B) ⊸ q(Is)).

q([enq(X)|Is]) ⟜ back([X|B]) ⊗ (back(B) ⊸ q(Is)).
q([deq(X)|Is]) ⟜ front([X|Xs]) ⊗ (front(Xs) ⊸ q(Is)).
q([]) ⟜ front([]) ⊗ back([]).
```

In this version, the dequeuing may borrow against future enqueue operations (see Exercise 15.2).

It is tempting to think we might use the linear context itself as a kind of queue, similar to the permutation program, but using cut '!' to avoid getting all solution. This actually does not work, since the linear context is maintained as a stack: most recently made assumptions are tried first.

15.7 Historical Notes

Resource management for linear logic programming was first considered by Hodas and Miller in the design of Lolli [7, 8]. The original design underestimated the importance of consumptive truth, which was later repaired by adding the slack indicator [5] and then the strict context [1, 2]. Primitive operations on the contexts are still quite expensive, which was addressed subsequently through so-called *tag frames* which implement the context management system presented here in an efficient way [6, 9].

An alternative approach to resource management is to use Boolean constraints to connect resources in different branches of the proof tree, developed by Harland and Pym [3, 4]. This is more general, because one is not committed to depth-first search, but also potentially more expensive.

An implementation of the linear logic programming language Lolli in Standard ML can be found at `http://www.cs.cmu.edu/~fp/lolli`. The code presented in this course, additional puzzles, a propositional theorem prover, and more example can be found in the distribution.

15.8 Exercises

Exercise 15.1 *Prove carefully that the* `perm` *predicate does implement permutation when invoked in the empty context. You will need to generalize this statement to account for intermediate states in the computation.*

Exercise 15.2 *Modify the stateful queue program so it fails if an element is dequeued before it is enqueued.*

Exercise 15.3 *Give an implementation of a double-ended queue in linear logic programming, where elements can be both enqueued and dequeued at both ends.*

Exercise 15.4 *Sometimes, linear logic appears excessively pedantic in that all resource must be used. In* affine logic *resources may be used at most once. Develop the connectives of affine logic, its asynchronous fragment, residuation, and resource management for affine logic. Discuss the simplifications in comparison with linear logic (if any).*

Exercise 15.5 *Philosophers have developed* relevance logic *in order to capture that in a proof of A implies B, some use of A should be made in the proof of B.* Strict logic *is a variant of relevance logic where we think of strict assumptions as resources which must be used at least once in a proof. Develop the connectives of strict logic, its asynchronous fragment, residuation, and resource management for strict logic. Discuss the simplifications in comparison with linear logic (if any).*

Exercise 15.6 *In the resource managment system we employed a relatively high-level logical system, without goal stack, failure continuation, or explicit unification. Extend the abstract machine which uses these mechanisms by adding resource management as given in this lecture.*

Because we can make linear or unrestricted assumptions in the course of search, not all information associated with a predicate symbol p is static, in the global program. This means the rule for atomic goals must change. You should fix the order in which assumptions are tried by using most recently made assumptions first and fall back on the static program when all dynamic possibilities have been exhausted.

15.9 References

[1] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.

[2] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[3] James Harland and David J. Pym. Resource distribution via Boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer Verlag LNCS 1249.

[4] James Harland and David J. Pym. Resource distribution via Boolean constraints. *ACM Transactions on Computational Logic*, 4(1):56–90, 2003.

[5] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[6] Joshua S. Hodas, Pablo López, Jeffrey Polakow, Lubomira Stoilova, and Ernesto Pimentel. A tag-frame system of resource management for proof search in linear-logic programming. In J. Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL'02)*, pages 167–182, Edinburgh, Scotland, September 2002. Springer Verlag LNCS 2471.

[7] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 32–42, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[8] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[9] Pablo López and Jeffrey Polakow. Implementing efficient resource management for linear logic programming. In Franz Baader and Andrei Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, pages 528–543, Montevideo, Uruguay, March 2005. Springer Verlag LNCS 3452.

15-819K: Logic Programming

Lecture 16

Substitution Semantics

Frank Pfenning

October 24, 2006

In this lecture we introduce a semantics with an explicit substitution, in preparation for presenting various program analyses later. The semantics will have the property that its goals appear exactly as in the original programs, with a separate substitution as part of the judgment. We also reviewed potential project areas during lecture, which is not represented in these notes.

16.1 Semantic Levels

When designing a program analysis we need to consider which level of semantic description is appropriate. This is relevant both for designing and proving the correctness of the analysis, which could be either simple or difficult, depending on our starting point. By the "level of semantic description" we mean here the spectrum from the logical semantics (in which we can only talk about truth), through one where subgoal order is explicit, to one with a failure continuation. An additional dimension is if a substitution is explicit in the semantics.

Which kind of semantics is appropriate, for example, for defining mode analysis? We would like to stay as high-level as possible, while still being able to express the property in question. Because mode analysis depends on subgoal order, one would expect to make subgoal order explicit. Moreover, since groundedness is a property of the substitution that is applied, we also should make a substitution explicit during computation. On the other hand, modes do not interact with backtracking, so we do not expect to need a failure continuation.

We take here a slight shortcut, using a semantics with an explicit substitution, but *not* a subgoal stack. Of course, it is possible to give such a

semantics as well. Omitting the subgoal stack has the advantage that we can relatively easily talk about the successful return of a predicate.

16.2 A Substitution Semantics

The semantics we give takes a goal G under a substitution τ. It produces a substitution θ with the invariant that $G\tau\theta\sigma$ for any grounding substitution σ. We define it on the fully residuated form, where for every predicate p there is exactly one clause, and this clause has the form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G$.

In the judgment $\tau \vdash G \;/\; \theta$ we maintain the invariant that $\mathrm{dom}(\tau) \supseteq \mathrm{FV}(G)$ and that $G\tau\theta \;\; true$ where we mean that there is a proof parametric in the remaining free variables. Moreover, θ substitutes only for logic variables X.

An important point[1] is that τ should substitute *exactly* for the originally quantified variables of G, and *not* for logic variables introduced during the computation. This is the role of θ which substitutes *only* for logic variables. The rule for atomic predicates is one place where this is important.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G) \in \Gamma \quad \mathbf{t}\tau/\mathbf{x} \vdash G \;/\; \theta}{\tau \vdash p(\mathbf{t}) \;/\; \theta}$$

We see that, indeed, the substitution on the premise accounts for all variables in G by the assumption of a closed normal form for programs.

The rule for conjunction presumes a subgoal order via the threading of θ_1, without using a subgoal stack.

$$\frac{\tau \vdash G_1 \;/\; \theta_1 \quad \tau[\theta_1] \vdash G_2 \;/\; \theta_2}{\tau \vdash G_1 \wedge G_2 \;/\; \theta_1\theta_2}$$

Here we have used a variant of the composition operator in order to maintain our invariant on the input substitution. $\tau[\theta_1]$ applies θ_1 to every element of τ, but does not extend it. That is,

$$(t_1/x_1, \ldots, t_n/x_n)[\theta] = (t_1[\theta]/x_1, \ldots, t_n[\theta]/x_n)$$

Truth is straightforward, as are the rules for disjunction and falsehood.

$$\frac{}{\tau \vdash \top \;/\; (\cdot)} \qquad \frac{\tau \vdash G_1 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta} \qquad \frac{\tau \vdash G_2 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta} \qquad \begin{array}{c}\text{no rule for} \\ \tau \vdash \bot \;/\; \_\end{array}$$

[1] I missed this point in lecture, which is why the system I gave did not work quite as well to prove the correctness of mode analysis.

The existential quantifier introduces a fresh logic variable X. This logic variable can somehow "escape" in that it may occur in the domain or co-domain θ. Intuitively, this makes sense because a logic variable that is not instantiated during the solution of G will remain after success.

$$\frac{X \notin \mathrm{FV}(\tau) \quad \tau, X/x \vdash G \mid \theta}{\tau \vdash \exists x.\, G \mid \theta}$$

Finally, equality reduces to unification.

$$\frac{\mathbf{t}\tau \doteq \mathbf{s}\tau \mid \theta}{\tau \vdash \mathbf{t} \doteq \mathbf{s} \mid \theta}$$

16.3 Correctness

The substitution semantics from the previous section is sound and complete in relation to the logical semantics of truth. First, the soundness. As remarked above, truth of a proposition with free variables is defined parametrically. That is, there must be one deduction with free variables every ground instance of which is true under the usual ground interpretation.

Theorem 16.1 *If $\tau \vdash G \mid \theta$ for $\mathrm{dom}(\tau) \supseteq \mathrm{FV}(G)$ then $G\tau\theta$ true.*

Proof: By induction on the deduction $\mathcal{D}$ of $\tau \vdash G \mid \theta$. For unification, we rely on soundness of unification. □

Completeness follows the usual pattern of lifting to deduction with free variables.

Theorem 16.2 *If $G\tau\sigma$ true where $\mathrm{dom}(\tau) \supseteq \mathrm{FV}(G)$ and $\mathrm{cod}(\sigma) = \emptyset$ then $\tau \vdash G \mid \theta$ and $\sigma = \theta\sigma'$ for some θ and σ'.*

Proof: By induction on the deduction of $G\tau\sigma$ θ. For unification we invoke the property that unification returns a most general unifier. □

16.4 An Asynchronous Substitution Semantics

Instead of giving substitution on goals for the residuated semantics, we can also give it directly on programs and goals if a normal form for programs is not desired or needed. There will be two judgments: $\tau \vdash A \mid \theta$ where A functions as a goal, and $\tau; A \ll P \mid \theta$ where A is formula under focus.

$$\frac{\tau \vdash A_1 \;/\; \theta_1 \quad \tau[\theta_1] \vdash A_2 \;/\; \theta_2}{\tau \vdash A_1 \wedge A_2 \;/\; \theta_1\theta_2} \qquad \frac{}{\tau \vdash \top \;/\; (\cdot)}$$

$$\text{omitted here}\quad \tau \vdash A_1 \supset A_2 \;/\; \_ \qquad \text{omitted here}\quad \tau \vdash \forall x.\, A \;/\; \_$$

$$\frac{\tau; A \ll P \;/\; \theta \quad A \in \Gamma}{\tau \vdash P \;/\; \theta}$$

$$\frac{\tau; A_1 \ll P \;/\; \theta}{\tau; A_1 \wedge A_2 \ll P \;/\; \theta} \qquad \frac{\tau; A_2 \ll P \;/\; \theta}{\tau; A_1 \wedge A_2 \ll P \;/\; \theta} \qquad \text{no rule for}\quad \tau; \top \ll P \;/\; \_$$

$$\frac{X \notin \mathrm{FV}(\tau) \quad (\tau, X/x); A \ll P \;/\; \theta}{\tau; \forall x.\, A \ll P \;/\; \theta} \qquad \frac{P'\tau \doteq P\tau \mid \theta}{\tau; P' \ll P \;/\; \theta}$$

$$\frac{\tau; A_1 \ll P \;/\; \theta_1 \quad \tau[\theta_1] \vdash A_2 \;/\; \theta_2}{\tau; A_2 \supset A_1 \ll P \;/\; \theta_1\theta_2}$$

We have not treated here implication and universal quantification as a goal. Implication is straightforward (see Exercise 16.1). Universal quantification in goals (which we have mostly avoided so far) creates difficulties for unification and is left to a future lecture.

The correctness theorem for this version of the semantics is left to Exercise 16.2.

16.5 Exercises

Exercise 16.1 *Extend the substitution semantics to permit dynamic assumptions* Γ *and goals of the form* $A_1 \supset A_2$*. Take care to account for the possibility that that dynamic assumptions may contain free variables.*

Exercise 16.2 *State and prove the correctness theorems for the asynchronous substitution semantics.*

15-819K: Logic Programming

Lecture 17

Mode Checking

Frank Pfenning

October 26, 2006

In this lecture we present *modes* to capture directionality in logic programs. Mode checking helps to catch programming errors and also allows for more efficient implementation. It is based on a form of *abstract interpretation* where substitutions are approximated by a two-element lattice: completely unknown terms, and terms known to be ground.

17.1 Modes

We have already informally used *modes* in the discussion of logic programs. For example, we can execute certain predicates in multiple directions, but this is impossible in other directions. We use modes + for *input*, − for *output* and ∗ for *bi-directional* arguments that are not designated as input or output.

We define the meaning as follows:

- Input arguments (+) must be ground when a predicate is invoked. It is an error if the mode analysis cannot establish this for a call to a predicate.
- Output arguments (−) must be ground when a predicate succeeds. It is an error if the mode analysis cannot establish this for the definition of a predicate.

From this we can deduce immediately:

- Input arguments (+) can be *assumed* to be ground when analyzing the definition of a predicate.

- Output arguments (`-`) can be *assumed* to be ground after a call to a predicate returns.

As an example, we return to our old program for addition.

```
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

First we check the mode

```
plus(+, +, -)
```

Consider the first clause, `plus(z, N, N)`. We are permitted to assume that the first two argument are ground, hence `N` is ground. We have to show the third argument, `N` is ground, which we just established. Therefore, the first clause is well-moded.

Looking at the head of the second clause, we may assume that `M` and `N` are ground. We have to eventually show the `s(P)` will be ground upon success. Now we analyze the body of the clause, `plus(M, N, P)`. This call is well-moded, because both `M` and `N` are known to be ground. Conversely, we may now assume that the output argument `P` is ground. Consequently `s(P)` is ground as required and the second clause is well-moded. Hence the definition of `plus` has the indicated mode.

I suggest you walk through showing that `plus` as has modes

```
plus(+, -, +)
plus(-, +, +)
```

which are two ways to calculate the difference of two numbers.

On the other hand, it does not have mode

```
plus(+, -, -)
```

because, for example, `plus(z, N, P)` succeeds without grounding either the second or third argument.

Modes are useful for a variety of purposes. First of all, they help to catch program errors, just like types. If a predicate does not satisfy an expected mode, it is likely a bug. This can happen fairly easily when names of free variables in clauses are mistyped. It is standard practice for Prolog compilers to produce a warning if a variable is used only once in a clause, under the assumption that it is a likely source of bugs. Unfortunately, this produces many false positives on correct programs. Mode errors are a much more reliable indication of bugs.

Secondly, modes can be used to produce more efficient code in a compiler, for example, by optimizing away occurs-checks (presuming you are interested in sound unification, as you should be), or by producing more efficient code for matching a goal against clause heads.

Thirdly, modes are helpful as a precondition for other analyses, such as termination analysis. For example, the `nat` predicate defined by

```
nat(s(N)) :- nat(N).
nat(z).
```

will terminate if the argument is ground, but diverge if the argument is a variable.

17.2 Semantic Levels

As laid out in the previous lecture, we need to consider which kind of semantics is appropriate for defining mode analysis? We would like to stay as high-level as possible, while still being able to express the property in question. Because mode analysis depends on subgoal order, one would expect to make subgoal order explicit. Moreover, since groundedness is a property related to the substitution that is applied, we also should make a substitution explicit during computation. On the other hand, modes do not interact with backtracking, so we don't expect to need a failure continuation.

We take here a slight shortcut, using a semantics with an explicit substitution, but *not* a subgoal stack. The resulting soundness property for mode analysis is not as strong as we may wish, as discussed in Exercise 17.1, but it is a bit easier to manage because it is easier to understand the concept of a successful return of a predicate call.

17.3 A Substitution Semantics

The semantics we gave in the previous lecture takes a goal G under a substitution τ. It produces a substitution θ with the invariant that $G\tau[\theta\sigma]$ for any grounding substitution σ. We define it on the fully residuated form, where for every predicate p there is exactly one clause, and this clause has the form $\forall \mathbf{x}, \mathbf{y}, \mathbf{z}.\, p(\mathbf{x}, \mathbf{y}, \mathbf{z}) \leftarrow G$ where $\mathbf{x}$ are the input arguments, $\mathbf{y}$ are the bi-directional arguments, and $\mathbf{z}$ are the output arguments of p. We have collected them into left-to-right form only for convenience, although this is probably also good programming practice. The rules are summarized in Figure 1.

$$\frac{\tau \vdash G_1 \;/\; \theta_1 \quad \tau[\theta_1] \vdash G_2 \;/\; \theta_2}{\tau \vdash G_1 \wedge G_2 \;/\; \theta_1\theta_2} \qquad \frac{}{\tau \vdash \top \;/\; (\cdot)}$$

$$\frac{(\forall \mathbf{x}, \mathbf{y}, \mathbf{z}.\, p(\mathbf{x}, \mathbf{y}, \mathbf{z}) \leftarrow G) \in \Gamma \quad \mathbf{t}\tau/\mathbf{x}, \mathbf{r}\tau/\mathbf{y}, \mathbf{s}\tau/\mathbf{z} \vdash G \;/\; \theta}{\tau \vdash p(\mathbf{t}, \mathbf{r}, \mathbf{s}) \;/\; \theta}$$

$$\frac{\tau \vdash G_1 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta} \qquad \frac{\tau \vdash G_2 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta} \qquad \begin{array}{c}\text{no rule for} \\ \tau \vdash \bot \;/\; \_\end{array}$$

$$\frac{X \notin \mathrm{FV}(\tau) \quad \tau, X/x \vdash G \;/\; \theta}{\tau \vdash \exists x.\, G \;/\; \theta} \qquad \frac{\mathbf{t}\tau \doteq \mathbf{s}\tau \mid \theta}{\tau \vdash \mathbf{t} \doteq \mathbf{s} \;/\; \theta}$$

Figure 1: Substitution Semantics

17.4 Abstract Substitutions

One common way of designing a program analysis is to construct and abstraction of a concrete domain involved in the operational semantics. Here, we abstract away from the substitution terms, tracking only if the terms are ground g or unknown u. They are related by an information ordering in the sense that u has no information and g is the most information we can have about a term. We write this in the form of a partial order which, in this case, is rather trivial.

We write $\mathsf{g} \leq \mathsf{u}$. If we are lower in this partial order we have more information; higher up we have less. This order is consistent with the interpretation of g and u as sets of terms: g is the set of ground terms, which is a subset of the set of all terms u.

Other forms of abstract interpretation, or a more detailed mode analysis, demands more complex orderings. We will see in the analysis which form of operations are required to be defined on the structure.

Abstract substitutions now have the form

$$\hat{\tau} ::= \cdot \mid \hat{\tau}, \mathsf{u}/x \mid \hat{\tau}, \mathsf{g}/x.$$

An abstract substitution $\hat{\tau}$ *approximates* an actual substitution τ if they have the same domain, and if whenever $\mathsf{g}/x \in \hat{\tau}$ it is indeed the case that $t/x \in \tau$ and $\mathrm{FV}(t) = \emptyset$. For variables marked as unknown in $\hat{\tau}$ there is no requirement placed on τ. We write $\hat{\tau} \preccurlyeq \tau$ if $\hat{\tau}$ approximates τ.

17.5 Mode Analysis Judgment

Now we abstract from the concrete operational semantics to one where we just carry abstract substitutions. The parallel nature of the operational and analysis rules leads to a mangeable proof of soundness of the analysis. Of course, completeness can not hold: there will always be programs that will respect modes at run-time, but fail the decidable judgment of well-modedness defined below (see Exercise 17.2).

At the top level we check each predicate separately. However, we assume that modes for all predicates are declared simultaneously, or at least that the modes of a predicate are defined before they are used in another predicate. The main judgment is

$$\hat{\tau} \vdash G \;/\; \hat{\sigma}$$

where $\mathrm{dom}(\tau) \supseteq \mathrm{FV}(G)$ and $\hat{\tau} \geq \hat{\sigma}$. The latter is interpreted pointwise, according to the partial order among the abstract elements. So $\hat{\tau}$ and $\hat{\sigma}$ have the same domain, and $\hat{\sigma}$ preserves all g/x in $\hat{\tau}$, but may transform some u/x into g/x.

When analyzing the definition of a predicate we are allowed to assume that all input arguments are ground and we have to show that upon success, the output arguments are ground. We do not assume or check anything about the bi-directional arguments.

$$\frac{\begin{array}{c}(\forall \mathbf{x}, \mathbf{y}, \mathbf{z}.\, p(\mathbf{x}, \mathbf{y}, \mathbf{z}) \leftarrow G) \in \Gamma \\ \mathbf{g}/\mathbf{x}, \mathbf{u}/\mathbf{y}, \mathbf{u}/\mathbf{z} \vdash G(\mathbf{x}, \mathbf{y}, \mathbf{z}) \;/\; (\mathbf{g}/\mathbf{x}, \_/\mathbf{y}, \mathbf{g}/\mathbf{z})\end{array}}{p \;\; wellmoded}$$

The judgment $\hat{\tau} \vdash \mathbf{t} \;\; ground$ checks that all terms in $\mathbf{t}$ are ground assuming the information given in $\hat{\tau}$.

$$\frac{\mathsf{g}/x \in \hat{\tau}}{\hat{\tau} \vdash x \;\; ground} \qquad \frac{\hat{\tau} \vdash \mathbf{t} \;\; ground}{\hat{\tau} \vdash f(\mathbf{t}) \;\; ground}$$

$$\frac{}{\hat{\tau} \vdash (\cdot) \;\; ground} \qquad \frac{\hat{\tau} \vdash t \;\; ground \quad \hat{\tau} \vdash \mathbf{t} \;\; ground}{\hat{\tau} \vdash (t, \mathbf{t}) \;\; ground}$$

Besides the abstraction, the gap to bridge between the operational and abstract semantics is only that fact in $\tau \vdash G \;/\; \theta$ the output θ is an increment: we apply it to τ to obtain the substitution under which G is true. $\hat{\sigma}$ on the other hand is a completed approximate substitution. This is reflected in the following property we show in the end.

If $\tau \vdash G \;/\; \theta$ and $\hat{\tau} \preccurlyeq \tau$ and $\hat{\tau} \vdash G \;/\; \hat{\sigma}$ then $\hat{\sigma} \preccurlyeq \tau[\theta]$.

Atoms. An atomic goal represents a procedure call. We therefore have to show that the input arguments are ground and we are permitted to assume that the output arguments will subsequently be ground.

$$\frac{\hat{\tau} \vdash \mathbf{t}\ \mathit{ground} \quad \hat{\tau} \vdash \mathbf{s} \;/\; \hat{\sigma}}{\hat{\tau} \vdash p(\mathbf{t}, \mathbf{r}, \mathbf{s}) \;/\; \hat{\sigma}}$$

The judgment $\hat{\tau} \vdash \mathbf{s} \;/\; \hat{\sigma}$ refines the information in $\hat{\tau}$ by noting that all variables in $\mathbf{s}$ can also be assumed ground.

$$\frac{}{(\hat{\tau}_1, \_/x, \hat{\tau}_2) \vdash x \;/\; (\hat{\tau}_1, \mathbf{g}/x, \hat{\tau}_2)} \qquad \frac{\hat{\tau} \vdash \mathbf{s} \;/\; \hat{\sigma}}{\hat{\tau} \vdash f(\mathbf{s}) \;/\; \hat{\sigma}}$$

$$\frac{}{\hat{\tau} \vdash (\cdot) \;/\; \hat{\tau}} \qquad \frac{\hat{\tau}_1 \vdash s \;/\; \hat{\tau}_2 \quad \hat{\tau}_2 \vdash \mathbf{s} \;/\; \hat{\tau}_3}{\hat{\tau}_1 \vdash (s, \mathbf{s}) \;/\; \tau_3}$$

Conjunction. Conjunctions are executed from left-to-right, so we propagate the mode information in the same order.

$$\frac{\hat{\tau}_1 \vdash G_1 \;/\; \hat{\tau}_2 \quad \hat{\tau}_2 \vdash G_2 \;/\; \hat{\tau}_3}{\hat{\tau}_1 \vdash G_1 \wedge G_2 \;/\; \hat{\tau}_3}$$

Truth. Truth does not affect any variables, so the modes are simply propagated unchanged.

$$\frac{}{\hat{\tau} \vdash \top \;/\; \hat{\tau}}$$

Disjunction. Disjunction represents an interesting challenge. For a goal $G_1 \vee G_2$ we do not know which subgoal will succeed. Therefore a variable

is only definitely known to be ground after execution of the disjunction, if it is known to be ground in both cases.

$$\frac{\hat{\tau} \vdash G_1 \sqcup \hat{\sigma}_1 \quad \hat{\tau} \vdash G_2 \sqcup \hat{\sigma}_2}{\hat{\tau} \vdash G_1 \vee G_2 \;/\; \hat{\sigma}_1 \sqcup \hat{\sigma}_2}$$

The least upper bound operation $\sqcup$ is applied to two abstract substitutions point-wise, and on abstract terms it is defined by

$$\begin{array}{lcl} \mathsf{g} \sqcup \mathsf{g} & = & \mathsf{g} \\ \mathsf{g} \sqcup \mathsf{u} & = & \mathsf{u} \\ \mathsf{u} \sqcup \mathsf{g} & = & \mathsf{u} \\ \mathsf{u} \sqcup \mathsf{u} & = & \mathsf{u} \end{array}$$

This is a common pattern in logic program analysis, where the least upper bound operations comes from the order on the abstract elements. To make sure that such a least upper bound always exist we generally stipulate that the order of abstract elements actually constitutes a *lattice* so that least upper bounds ($\sqcup$) and greatest lower bounds ($\sqcap$) of finite sets always exist. As a special case, the least upper bound of the empty set is the bottom element of the lattice, usually denoted by $\bot$ or 0. We use the latter because $\bot$ is already employed with its logical meaning. Here, the bottom element is g for an individual element, and $\mathbf{g}/\mathbf{x}$ for a substitution.

Falsehood. In the operational semantics there is no rule for proving falsehood. In the mode analysis, however, we need a rule for handling falsehood, since analysis should not fail unless there is a mode error. Recall that we need for the output of mode analysis to approximate the output substitution $\tau[\theta]$ if $\tau \vdash G \;/\; \theta$. But $G = \bot$ can never succeed, so this requirement is vacuous. Consequently, is it safe to pick the bottom element of the lattice.

$$\frac{\mathrm{dom}(\hat{\tau}) = \mathbf{x}}{\hat{\tau} \vdash \bot \;/\; (\mathbf{g}/\mathbf{x})}$$

Existential. Solving an existential creates a new logic variable. This is still a free variable, so we mark its value as not known to be ground.

$$\frac{(\hat{\tau}, \mathsf{u}/x) \vdash G \;/\; (\hat{\sigma}, \_/x)}{\hat{\tau} \vdash \exists x.\, G \;/\; \hat{\sigma}}$$

Since there is no requirement that existential variables eventually become ground, we do not care what is known about the substitution term for x upon the successful completion of G.

Equality. When encountering an equality we need to descend into the terms, abstractly replaying unification, to approximate the resulting substitution. We therefore distinguish various cases. Keep in mind that analysis should always succeed, even if unification fails at runtime, which gives us more cases to deal with than one would initially expect. We write $\hat{\tau} + \mathsf{g}/x$ for the result of setting the definition for x in $\hat{\tau}$ to g.

$$\frac{\hat{\tau} \vdash s \; \mathit{ground}}{\hat{\tau} \vdash x \doteq s \;/\; \hat{\tau} + \mathsf{g}/x} \qquad \frac{\mathsf{g}/x \in \hat{\tau} \quad \hat{\tau} \not\vdash s \; \mathit{ground} \quad \hat{\tau} \vdash s \;/\; \hat{\sigma}}{\hat{\tau} \vdash x \doteq s \;/\; \hat{\sigma}} \qquad \frac{\mathsf{u}/x \in \hat{\tau} \quad \hat{\tau} \not\vdash s \; \mathit{ground}}{\hat{\tau} \vdash x \doteq s \;/\; \hat{\tau}}$$

$$\frac{\hat{\tau} \vdash t \; \mathit{ground}}{\hat{\tau} \vdash t \doteq y \;/\; \hat{\tau} + \mathsf{g}/y} \qquad \frac{\mathsf{g}/y \in \hat{\tau} \quad \hat{\tau} \not\vdash t \; \mathit{ground} \quad \hat{\tau} \vdash t \;/\; \hat{\sigma}}{\hat{\tau} \vdash t \doteq y \;/\; \hat{\sigma}} \qquad \frac{\mathsf{u}/y \in \hat{\tau} \quad \hat{\tau} \not\vdash t \; \mathit{ground}}{\hat{\tau} \vdash t \doteq y \;/\; \hat{\tau}}$$

$$\frac{\hat{\tau} \vdash \mathbf{t} \doteq \mathbf{s} \;/\; \hat{\sigma}}{\hat{\tau} \vdash f(\mathbf{t}) \doteq f(\mathbf{s}) \;/\; \hat{\sigma}} \qquad \frac{f \neq g \quad \mathrm{dom}(\hat{\tau}) = \mathbf{x}}{\hat{\tau} \vdash f(\mathbf{t}) \doteq g(\mathbf{s}) \;/\; (\mathbf{g}/\mathbf{x})}$$

$$\frac{\hat{\tau}_1 \vdash t \doteq s \;/\; \hat{\tau}_2 \quad \hat{\tau}_2 \vdash \mathbf{t} \doteq \mathbf{s} \;/\; \hat{\tau}_3}{\hat{\tau}_1 \vdash (t, \mathbf{t}) \doteq (s, \mathbf{s}) \;/\; \hat{\tau}_3} \qquad \frac{}{\hat{\tau} \vdash (\cdot) \doteq (\cdot) \;/\; \hat{\tau}}$$

$$\frac{\mathrm{dom}(\hat{\tau}) = \mathbf{x}}{\hat{\tau} \vdash (\cdot) \doteq (s, \mathbf{s}) \;/\; (\mathbf{g}/\mathbf{x})} \qquad \frac{\mathrm{dom}(\hat{\tau}) = \mathbf{x}}{\hat{\tau} \vdash (t, \mathbf{t}) \doteq (\cdot) \;/\; (\mathbf{g}/\mathbf{x})}$$

The first and second lines overlap in the sense that for some equations, more than one rule applies. However, the answer is the same in either case. We could resolve the ambiguity by requiring, for example, that in the second line t is not a variable, that is, of the form $f(\mathbf{t})$.

17.6 Soundness

Next we have to prove soundness of the analysis. First we need a couple of lemmas regarding the term-level judgments. One can view these as encoding what happens when an approximate substitution is applied, so we refer to them as the first and second approximate substitution lemma. To see how they arise you might analyze the soundness proof below first.

Lemma 17.1 *If $\hat{\tau} \vdash t$ ground and $\hat{\tau} \preccurlyeq \tau$ and then $t\tau$ ground.*

Proof: By induction on the structure of $\mathcal{D}$ of $\hat{\tau} \vdash t \; \mathit{ground}$. □

Lemma 17.2 *If $\hat{\tau} \vdash s \;/\; \hat{\sigma}$ and $\hat{\tau} \preccurlyeq \tau$ and $s\tau[\theta]$ ground then $\hat{\sigma} \preccurlyeq \tau[\theta]$.*

Proof: By induction on the structure of $\mathcal{D}$ of $\hat{\tau} \vdash s \;/\; \hat{\sigma}$. □

Theorem 17.3 *If $\tau \vdash G \;/\; \theta$ and $\hat{\tau} \preccurlyeq \tau$ and $\hat{\tau} \vdash G \;/\; \hat{\sigma}$ then $\hat{\sigma} \preccurlyeq \tau[\theta]$.*

Proof: By induction on the structure of $\mathcal{D}$ of $\tau \vdash G \;/\; \theta$, applying inversion to the given mode derivation in each case.

Case: $\mathcal{D} = \dfrac{(\forall \mathbf{x}, \mathbf{y}, \mathbf{z}.\, p(\mathbf{x}, \mathbf{y}, \mathbf{z}) \leftarrow G) \in \Gamma \quad (\mathbf{t}^+\tau/\mathbf{x}, \mathbf{r}^*\tau/\mathbf{y}, \mathbf{s}^-\tau/\mathbf{z}) \vdash G \;/\; \theta}{\tau \vdash p(\mathbf{t}^+, \mathbf{r}^*, \mathbf{s}^-) \;/\; \theta}$.

$\hat{\tau} \vdash p(\mathbf{t}^+, \mathbf{r}^*, \mathbf{s}^-) \;/\; \hat{\sigma}$	Assumption
$\hat{\tau} \vdash \mathbf{t}^+$ *ground* and	
$\hat{\tau} \vdash \mathbf{s}^- \;/\; \hat{\sigma}$	By inversion
$\hat{\tau} \preccurlyeq \tau$	Assumption
$\mathbf{t}^+\tau$ *ground*	By approx. subst. lemma
$(\mathbf{g}/\mathbf{x}, \mathbf{u}/\mathbf{y}, \mathbf{u}/\mathbf{z}) \preccurlyeq (\mathbf{t}^+\tau/\mathbf{x}, \mathbf{r}^*\tau/\mathbf{y}, \mathbf{s}^-\tau/\mathbf{z})$	From previous line
p *wellmoded*	Assumption
$\mathbf{g}/\mathbf{x}, \mathbf{u}/\mathbf{y}, \mathbf{u}/\mathbf{z} \vdash G \;/\; (\mathbf{g}/\mathbf{x}, \_/\mathbf{y}, \mathbf{g}/\mathbf{z})$	By inversion
$(\mathbf{g}/\mathbf{x}, \_/\mathbf{y}, \mathbf{g}/\mathbf{z}) \preccurlyeq (\mathbf{t}^+\tau\theta/\mathbf{x}, \mathbf{r}^*\tau\theta/\mathbf{y}, \mathbf{s}^-\tau\theta/\mathbf{z})$	By ind.hyp.
$\mathbf{s}^-\tau\theta$ *ground*	By defn. of $\preccurlyeq$
$\hat{\sigma} \preccurlyeq \tau\theta$	By approx. subst. lemma

Case: $\mathcal{D} = \dfrac{\tau \vdash G_1 \;/\; \theta_1 \quad \tau\theta_1 \vdash G_2 \;/\; \theta_2}{\tau \vdash G_1 \wedge G_2 \;/\; \theta_1\theta_2}$.

$\hat{\tau} \vdash G_1 \wedge G_2 \;/\; \hat{\sigma}_2$	Assumption
$\hat{\tau} \vdash G_1 \;/\; \hat{\sigma}_1$ and	
$\hat{\sigma}_1 \vdash G_2 \;/\; \hat{\sigma}_2$ for some $\hat{\sigma}_1$	By inversion
$\hat{\tau} \preccurlyeq \tau$	Assumption
$\hat{\sigma}_1 \preccurlyeq \tau[\theta_1]$	By ind.hyp.
$\hat{\sigma}_2 \preccurlyeq (\tau[\theta_1])[\theta_2]$	By ind.hyp.
$\hat{\sigma}_2 \preccurlyeq \tau[\theta_1\theta_2]$	By assoc of composition

Case: $\mathcal{D} = \dfrac{}{\tau \vdash \top \;/\; (\cdot)}$.

$\hat{\tau} \vdash \top \;/\; \hat{\sigma}$	Assumption
$\hat{\sigma} = \hat{\tau}$	By inversion
$\hat{\tau} \preccurlyeq \tau$	Assumption
$\hat{\sigma} \preccurlyeq \tau[\cdot]$	By identity of $(\cdot)$

Case: $\mathcal{D} = \dfrac{\tau \vdash G_1 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta}$.

$\hat{\tau} \vdash G_1 \vee G_2 \;/\; \hat{\sigma}$	Assumption
$\hat{\tau} \vdash G_1 \;/\; \hat{\sigma}_1$ and	
$\hat{\tau} \vdash G_2 \;/\; \hat{\sigma}_2$ for some $\hat{\sigma}_1$, $\hat{\sigma}_2$ with $\hat{\sigma} = \hat{\sigma}_1 \sqcup \hat{\sigma}_2$	By inversion
$\hat{\tau} \preccurlyeq \tau$	Assumption
$\hat{\sigma}_1 \preccurlyeq \tau[\theta]$	By ind.hyp.
$\hat{\sigma}_1 \sqcup \hat{\sigma}_2 \preccurlyeq \tau[\theta]$	By property of least upper bound

Case: $\mathcal{D} = \dfrac{\tau \vdash G_2 \;/\; \theta}{\tau \vdash G_1 \vee G_2 \;/\; \theta}$. Symmetric to the previous case.

Case: $\tau \vdash \bot \;/\; \theta$. There is no rule to conclude such a judgment. Therefore the property holds vacuously.

Case: $\mathcal{D} = \dfrac{\tau, X/x \vdash G \;/\; \theta \quad X \notin \mathrm{FV}(\tau)}{\tau \vdash \exists x.\, G \;/\; \theta}$.

$\hat{\tau} \vdash \exists x.\, G \;/\; \hat{\sigma}$	Assumption
$\hat{\tau}, \mathsf{u}/x \vdash G \;/\; (\hat{\sigma}, \_/x)$	By inversion
$\hat{\tau} \preccurlyeq \tau$	Assumption
$(\hat{\tau}, \mathsf{u}/x) \preccurlyeq (\tau, X/x)$	By defn. of $\preccurlyeq$
$(\hat{\sigma}, \_/x) \preccurlyeq (\tau, X/x)[\theta]$	By ind.hyp.
$\hat{\sigma} \preccurlyeq \tau[\theta]$	By prop. of subst. and approx.

Case: $\mathcal{D} = \dfrac{\mathbf{t}\tau \doteq \mathbf{s}\tau \mid \theta}{\tau \vdash \mathbf{t} \doteq \mathbf{s} \;/\; \theta}$. This case is left to the reader (see Exercise 17.3).

□

17.7 Exercises

Exercise 17.1 *One problem with well-modedness in this lecture is that we only prove that if a well-moded query succeeds then the output will be ground. A stronger property would be that during the execution of the program, every goal and subgoal we consider will be well-moded. However, this requires a transition semantics and a different soundness proof.*

Write a suitable operational semantics and prove soundness of mode checking in the sense sketched above. This is a kind of mode preservation theorem, analogous to a type preservation theorem.

Exercise 17.2 *Give a program that respects modes at run-time in the sense that*

- *input arguments (+) are always ground when a predicate is invoked, and*
- *output arguments (−) are always ground when a predicate succeeds,*

and yet is not well-moded according to our analysis.

Exercise 17.3 *Complete the proof of soundness of mode analysis by giving the case for unification. If you need a new lemma in addition to the approximate substitution lemmas, carefully formulate and prove them.*

15-819K: Logic Programming

Lecture 18

Proof Terms

Frank Pfenning

October 31, 2006

In this lecture we will substantiate an earlier claim that logic programming not only permits the representation of specifications and the implementation of algorithm, but also the realization of proofs of correctness of algorithms. We will do so by first showing how *deductions* can be represented as terms, so-called *proof terms*. We also discuss the problems of checking the validity of deductions, which amounts to type checking the terms that represent them.

18.1 Deductions as Terms

In logic programming we think of computation as proof search. However, so far search only returns either success together with an answer substitution, fails, or diverges. In the case of success it is reasonable to expect that the logic programming engine could also return a deduction of the instantiated goal. But this raises the question of how to represent deductions. In the traditional logic programming literature we will find ideas such as a list of the rules that have been applied to solve a goal. There is, however, a much better answer. We can think of an inference rule as a *function* which takes deductions of the premisses to a deduction of the conclusion. Such a function is a constructor for proof terms. For example, when interpreting

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

we extract two constructors

$$\begin{array}{lcl} \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)). \end{array}$$

The only unusual thing about these constructors is their type. The type of pz, for example, is a *proposition*.

The idea that *propositions* can be *types* is a crucial observation of the Curry-Howard isomorphism in functional programming that also identifies proofs (of a proposition) with programs (of the corresponding type). Here, the correspondence of propositions with types is still perfect, but proofs are *not* programs (which, instead, or are given by inference rules).

As an example of how a complete proof is interpreted as a term, consider the computation of $2 + 2 = 4$.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{plus}(1,2,3)}\,\mathsf{ps}}{\mathsf{plus}(2,2,4)}\,\mathsf{ps}$$

Here we have abbreviated $\mathsf{z} = 0, \mathsf{s}(\mathsf{z}) = 1, \ldots$. This deduction becomes the very simple term

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})).$$

Our typing judgment should be such that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2,2,4)$$

so that a proposition acts as the type of its proofs.

As a slightly more complex example, consider multiplication

$$\dfrac{}{\mathsf{times}(\mathsf{z},N,\mathsf{z})}\,\mathsf{tz} \qquad \dfrac{\mathsf{times}(M,N,P) \quad \mathsf{plus}(P,N,Q)}{\mathsf{times}(\mathsf{s}(M),N,Q)}\,\mathsf{ts}$$

which yields the following constructors

$$\begin{array}{lcl} \mathsf{tz} & : & \mathsf{times}(\mathsf{z},N,\mathsf{z}) \\ \mathsf{ts} & : & \mathsf{times}(M,N,P), \mathsf{times}(P,N,Q) \rightarrow \mathsf{times}(M,N,Q). \end{array}$$

Now the deduction that $2 * 2 = 4$, namely

$$\dfrac{\dfrac{\dfrac{}{\mathsf{times}(0,2,0)}\,\mathsf{tz} \quad \dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{times}(1,2,2)}\,\mathsf{ts} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{plus}(1,2,3)}\,\mathsf{ps}}{\mathsf{plus}(2,2,4)}\,\mathsf{ps}}{\mathsf{times}(2,2,4)}\,\mathsf{ts}$$

becomes the term

$$\mathsf{ts}(\mathsf{ts}(\mathsf{tz}, \mathsf{pz}), \mathsf{ps}(\mathsf{ps}(\mathsf{pz}))) : \mathsf{times}(2, 2, 4).$$

The tree structure of the deduction is reflected in the corresponding tree structure of the term.

18.2 Indexed Types

Looking at our example signature,

$$\begin{array}{lcl} \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \\ \\ \mathsf{tz} & : & \mathsf{times}(\mathsf{z}, N, \mathsf{z}) \\ \mathsf{ts} & : & \mathsf{times}(M, N, P), \mathsf{times}(P, N, Q) \rightarrow \mathsf{times}(M, N, Q) \end{array}$$

we can observe a new phenomenon. The types of our constructors contain *terms*, both constants (such as z) and variables (such as M, N, or P). We say that plus is a *type family indexed* by terms. In general, under the propositions-as-types interpretation, predicates are interpreted type families indexed by terms.

The free variables in the declarations are interpreted *schematically*, just like in inference rules. So pz is really a family of constants, indexed by N. This has some interesting consequences. For example, we found earlier that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 2, 4).$$

However, we can also check that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 3, 5).$$

In fact, our type system will admit a most general type:

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(\mathsf{s}(\mathsf{s}(\mathsf{z})), N, \mathsf{s}(\mathsf{s}(N))).$$

This schematic type captures all types of the term on the left, because any type for ps(ps(pz)) is an instance of plus(s(s(z)), N, s(s(N))).

18.3 Typing Rules

In order to write down the typing rules, it is convenient to make quantification over schematic variables in a indexed type declaration explicit. We

write $\forall x{:}\tau$ for quantification over a single variable, and following our general notational convention, $\forall \mathbf{x}{:}\boldsymbol{\tau}$ for a sequence of quantifiers. We have already introduced quantified propositions, so we emphasize its role as quantifying over the schematic variables of a proposition viewed as a type. The example of addition would be written as

$$\begin{array}{lcl} \mathsf{z} & : & \mathsf{nat} \\ \mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} \end{array}$$

$$\begin{array}{lcl} \mathsf{pz} & : & \forall N{:}\mathsf{nat}.\, \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \forall M{:}\mathsf{nat}.\, \forall N{:}\mathsf{nat}.\, \forall P{:}\mathsf{nat}.\, \mathsf{plus}(M, N, P) \to \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \end{array}$$

We call such explicitly quantified types *dependent types*. Unlike other formulations (for example, in the LF logical framework), but similarly to our treatment of polymorphism, the quantifiers do not affect the term language. We write '$\forall$' to emphasize the logical reading of the quantifiers; in a fully dependent type theory they would be written as 'Π'.

There are a many of similarities between polymorphic and dependent types. We will see that in addition to the notation, also the typing rules are analogous. Nevertheless, they are different concepts: polymorphism quantifies over types, while dependency quantifies over terms. We review the rule for polymorphic typing.

$$\frac{\begin{array}{l}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \qquad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

Recall that $\hat{\theta}$ is a substitution of types for type variables, and that Δ contains declarations $x{:}\tau$ as well as $\alpha\ \mathit{type}$.

The rule for dependent types is analogous, using ordinary substitutions instead of type substitution.

$$\frac{\begin{array}{l}\mathrm{dom}(\theta) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \to P \in \Sigma \qquad \Delta \vdash \mathbf{t} : \mathbf{Q}\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P\theta}$$

We have written P and $\mathbf{Q}$ instead of τ and $\boldsymbol{\sigma}$ to emphasize the interpretation of the types as propositions.

If we require θ to be a ground substitution (that is, $\mathrm{cod}(\theta) = \emptyset$), then we can use this rule to determine ground typings such as

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 2, 4).$$

If we allow free variables, that is, $\mathrm{cod}(\theta) = \mathrm{dom}(\Delta)$, then we can write out schematic typings, such as

$$n{:}\mathsf{nat} \vdash \mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(\mathsf{s}(\mathsf{s}(\mathsf{z})), n, \mathsf{s}(\mathsf{s}(n))).$$

In either case we want the substitution to be well-typed. In the presence of dependent types, the formalization of this would lead us a bit far afield, so we can think of it just as before: we always substitute a term of type τ for a variable of type τ.

18.4 Type Checking

Ordinarily in logic programming a query is simply a proposition and the result is an answer substitution for its free variables. When we have proof terms we can also ask if a given term constitutes a proof of a given proposition. We might write this as

```
?- t : P.
```

where t is a term representing a purported proof and P is a goal proposition. From the typing rule we can see that type-checking comes down to unification. We can make this more explicit by rewriting the rule:

$$\frac{\begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \rightarrow P' \in \Sigma \qquad P'\rho \doteq P \mid \theta \quad \Delta \vdash \mathbf{t} : \mathbf{Q}\rho\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P}$$

Here ρ is a renaming substituting generating fresh logic variables for the bound variables $\mathbf{x}$.

Because a constant has at most one declaration in a signature, and unification returns a unique most general unifier, the type-checking process is entirely deterministic and will always either fail (in which case there is no type) or succeed. We can even leave P as a variable and obtain the most general type for a given term t.

Checking that a given term represents a valid proof can be useful in a number of situations. Some practical scenarios where this has been applied is *proof-carrying code* and *proof-carrying authorization*. Proof-carrying code is the idea that we can equip a piece of mobile code with a proof that it is safe to execute. A code recipient can check the validity of the proof against the code and then run the code without further run-time checks. Proof-carrying authorization is a similar idea, except that the proof is used to

convince a resource monitor that a client is authorized for access. Please see the section on historical notes for some references on these applications of logic programming.

18.5 Proof Search

Coming back to proof search: we would like to instrument our interpreter so it returns a proof term (as well as an answer substitution) when it succeeds. But the exact rule for type-checking with a slightly different interpretation on modes, will serve that purpose.

$$\frac{\begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \rightarrow P' \in \Sigma \qquad P'\rho \doteq P \mid \theta \quad \Delta \vdash \mathbf{t} : \mathbf{Q}\rho\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P}$$

Above, we thought of $c(\mathbf{t})$ and therefore $\mathbf{t}$ as given input, so this was a rule for type-checking. Now we think of $\mathbf{t}$ as an output, produced by proof search for the premiss, which then allows us to construct $c(\mathbf{t})$ as an output in the conclusion. Now the rule is non-deterministic since we do not know which rule for a given atomic predicate to apply, but for a given proof we will be able to construct a proof term as an output.

We have not addressed here if ordinary untyped unification will be sufficient for program execution (or, indeed, type-checking), or if unification needs to be changed in order to take typing into account. After a considerable amount of technical work, we were able to show in the case of polymorphism that function symbols needed to be type preserving and clause heads parametric for untyped unification to suffice. If we explicitly stratify our language so that in a declaration $c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \rightarrow P'$ all the types $\boldsymbol{\tau}$ have no variables then the property still holds for well-typed queries; otherwise it may not (see Exercise 18.1).

18.6 Meta-Theoretic Proofs as Relations

We now take a further step, fully identifying types with propositions. This means that quantifiers in clauses can now range over *deductions*, and we can specify relations between deductions. Deductions have now become first-class.

There are several uses for first-class deductions. One is that we can now implement theorem provers or decision procedures in a way that intrinsically guarantees the validity of generated proof objects.

Another application is the implementation of proofs *about* the predicates that make up logic programs. To illustrate this, we consider the proof that the sum of two even numbers is even. We review the definitions:

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{ez} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{ess}$$

so that the type declarations for proof constructors are

$$\begin{array}{lcl} \mathsf{ez} & : & \mathsf{even}(\mathsf{z}) \\ \mathsf{ess} & : & \mathsf{even}(N) \rightarrow \mathsf{even}(\mathsf{s}(\mathsf{s}(N))) \\ \\ \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \end{array}$$

Theorem 18.1 *For any m, n, and p, if* $\mathsf{even}(m)$, $\mathsf{even}(n)$, *and* $\mathsf{plus}(m, n, p)$ *then* $\mathsf{even}(p)$.

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $\mathsf{even}(m)$.

Case: $\mathcal{D} = \dfrac{}{\mathsf{even}(\mathsf{z})}$ where $m = \mathsf{z}$.

$\mathsf{even}(n)$	Assumption
$\mathsf{plus}(\mathsf{z}, n, p)$	Assumption
$n = p$	By inversion
$\mathsf{even}(p)$	Since $n = p$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(m')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

$\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$	Assumption
$\mathsf{plus}(\mathsf{s}(m'), n, p')$ with $p = \mathsf{s}(p')$	By inversion
$\mathsf{plus}(m', n, p'')$ with $p' = \mathsf{s}(p'')$	By inversion
$\mathsf{even}(p'')$	By ind. hyp.
$\mathsf{even}(\mathsf{s}(\mathsf{s}(p'')))$	By rule

□

The theorem and its proof involves four deductions:

$$\underset{\mathsf{even}(m)}{\mathcal{D}} \qquad \underset{\mathsf{even}(n)}{\mathcal{E}} \qquad \underset{\mathsf{plus}(m,n,p)}{\mathcal{F}} \qquad \underset{\mathsf{even}(p)}{\mathcal{G}}$$

The theorem states that for any derivations $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ there exists a deduction $\mathcal{G}$. Using our newfound notation for proof terms we can write this as

> *For every m, n, and p, and for every $D : \mathsf{even}(m)$, $E : \mathsf{even}(n)$, and $F : \mathsf{plus}(m, n, p)$ there exists $G : \mathsf{even}(p)$.*

If this course were about functional programming, we would ensure that this theorem holds by exhibing a *total function*

$$\mathsf{eee} : \mathsf{even}(M) \to \mathsf{even}(N) \to \mathsf{plus}(M, N, P) \to \mathsf{even}(P).$$

It is important that the function be total so that it is guaranteed to generate an output deduction of $\mathsf{even}(P)$ for any combination of input deductions, thereby witnessing its truth.

In logic programming, such functions are not at our disposal. But we can represent the same information as a *total relation*

$$\mathsf{eee} : \mathsf{even}(M), \mathsf{even}(N), \mathsf{plus}(M, N, P), \mathsf{even}(P) \to o.$$

The predicate symbol eee represents a four-place relation between deductions, where we consider the first three deductions as inputs and the last one as an output.

In the next lecture we consider in some details what is required to *verify* that this relation represents a meta-theoretic proof of the property that the sum of two even number is even. Before we get to that, let us examine how our careful, but informal proof is translated into a relation. We will try to construct clauses for

$$\mathsf{eee}(D, E, F, G)$$

where $D : \mathsf{even}(M)$, $E : \mathsf{even}(N)$, $F : \mathsf{plus}(M, N, P)$, and $G : \mathsf{even}(P)$. We repeat the proof, analyzing the structure of D, E, F, and G. We highlight the incremental construction of clauses for eee in interspersed boxes.

Case: $D = \dfrac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{ez}$ where $m = \mathsf{z}$.

At this point we start to construct a clause

$$\mathsf{eee}(\mathsf{ez}, E, F, G)$$

because $D = \mathsf{ez}$, and we do not yet know E, F, or G.

$E : \mathsf{even}(n)$ Assumption
$F : \mathsf{plus}(\mathsf{z}, n, p)$ Assumption
$F = \mathsf{pz}$ and $n = p$ By inversion

At this point we have some more information, namely $F = \mathsf{pz}$. So the partially constructed clause now is

$$\mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, G).$$

$G = E : \mathsf{even}(p)$ Since $n = p$

Now we see that the output G is equal to the second input E.

$$\mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, E)$$

This completes the construction of this clause. The second argument E is not analyzed and simply returned as G.

Case: $D = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(m')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}\ \mathsf{ess}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

The partially constructed second clause now looks like

$$\mathsf{eee}(\mathsf{ess}(D'), E, F, G).$$

$F : \mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$ Assumption
$F = \mathsf{ps}(F')$ where $F' : \mathsf{plus}(\mathsf{s}(m'), n, p')$ with $p = \mathsf{s}(p')$ By inversion

Now we have

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(F'), G)$$

replacing F above by $\mathsf{ps}(F')$.

$F' = \mathsf{ps}(F'')$ where $F'' : \mathsf{plus}(m', n, p'')$ with $p' = \mathsf{s}(p'')$ By inversion

In this step, the third argument has been even further refined to $\mathsf{ps}(\mathsf{ps}(F''))$ which yields

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), G).$$

$G' : \mathsf{even}(p'')$ By ind. hyp. on D', E, and F''

An appeal to the induction hypothesis corresponds to a recursive call in the definition of eee.

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), G) \leftarrow \mathsf{eee}(D', E, F'', G')$$

The first three arguments of the recursive call correspond to the deductions on which the induction hypothesis is applied, the fourth argument is the returned deduction G'. The question of how we construct the G still remains.

$G = \mathsf{ess}(G') : \mathsf{even}(\mathsf{s}(\mathsf{s}(p'')))$ By rule ess applied to G'

Now we can fill in the last missing piece by incorporating the definition of G.

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), \mathsf{ess}(G')) \leftarrow \mathsf{eee}(D', E, F'', G')$$

In summary, the meta-theoretic proof is represented as the relation eee shown below. We have named the rules defining eee for consistency, even though it seems unlikely we will want to refer to these rules by name.

$$\begin{array}{lcl} \mathsf{eee} & : & \mathsf{even}(M), \mathsf{even}(N), \mathsf{plus}(M, N, P), \mathsf{even}(P) \rightarrow o. \\ \\ \mathsf{eeez} & : & \mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, E). \\ \mathsf{eeess} & : & \mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), \mathsf{ess}(G')) \leftarrow \mathsf{eee}(D', E, F'', G'). \end{array}$$

Each case in the definition of eee corresponds to a case in the inductive proof, a recursive call corresponds to an appeal to the induction hypothesis. A constructed term on an input argument of the clause head corresponds to a case split on the induction variable or an appeal to inversion. A constructed term in an output position of the clause head corresponds to a rule application to generate the desired deduction.

It is remarkable how compact the representation of the informal proof has become: just one line declaring the relation and two lines defining the relation. This is in contrast to the informal proof which took up 11 lines.

18.7 Verifying Proofs of Meta-Theorems

In the previous section we showed how to represent a proof of a theorem about deductions as a relation between proofs. But what does it take to verify that a given relation indeed represents a meta-theoretic proof of a proposed theorem? A full treatment of this question is beyond the scope of this lecture (and probably this course), but meta-logical frameworks such as Twelf can indeed verify this. Twelf decomposes this checking into multiple steps, each making its own contribution to the overall verification.

1. Type checking. This guarantees that if $\mathsf{eee}(D, E, F, G)$ for deductions D, E, F, and G is claimed, all of these are valid deductions. In particular, the output G will be acceptable evidence that P is even because $G : \mathsf{even}(P)$.

2. Mode checking. The mode $\mathsf{eee}(+, +, +, -)$ guarantees that if the inputs are all complete (ground) deductions and the query succeeds, then the output is also a complete (ground) deduction. This is important because a free variable in a proof term represents an unproven subgoal. Such deductions would be insufficient as evidence that P is indeed even.

3. Totality checking. If type checking and mode checking succeeds, we know that P is even if a query $\mathsf{eee}(D, E, F, G)$ succeeds for ground D, E, F and free variable G. All that remains is to show that *all* such queries succeed. We decompose this into two properties.

   (a) Progress checking. For any given combination of input deductions D, E, and F there is an applicable clause and we either succeed or at least can proceed to a subgoal. Because of this, $\mathsf{eee}(D, E, F, G)$ can never fail.

   (b) Termination checking. Any sequence of recursive calls will terminate. Since queries can not fail (by progress) the only remaining possibility is that they succeed, which is what we needed to verify.

We have already extensively discussed type checking and mode checking. In the next lecture we sketch progress checking. For termination checking we refer to the literature in the historical notes below.

18.8 Polymorphism and Dependency

Because of the similarity of polymorphic and dependent types, and because both have different uses, it is tempting to combine the two in a single language. This is indeed possible. For example, if we would like to index a polymorphic list with its length we could define

```
nat : type.
z : nat.
s : nat -> nat.

list : type,nat -> type.
nil : list(A,z).
cons : A,list(A,N) -> list(A,s(N)).
```

In the setting of functional programming, such combinations have been thoroughly investigated, for examples, as fragments of the Calculus of Constructions. In the setting here we are not aware of a thorough study of either type checking or unification. A tricky issue seems to be how much type information must be carried at run-time, during unification, especially if a type variable can be instantiated by a dependent type.

18.9 Historical Notes

The notion of proof term originates in the so-called Curry-Howard isomorphism, which was noted for combinatory logic by Curry and Feys in 1956 [8] and for natural deduction by Howard in 1969 [10]. The emphasis in this work is on functional computation, by combinatory reduction or β-reduction, respectively. The presentation here is much closer to the LF logical framework [9] in which only canonical forms are relevant, and which is entirely based on dependent types. We applied one main simplification: because we restricted ourselves to the Horn fragment of logic, proof terms contain no λ-abstractions. This in turn allows us to omit Π-quantifiers without any loss of decidability of type checking, since type inference can be achieved just by ordinary unification.

The combination of polymorphism and dependency is present in the Calculus of Constructions [6]. Various fragments were later analyzed in detail by Barendregt [2], including the combination of polymorphism with dependent types. A modern version of the Calculus of Constructions is the basis for the Coq theorem proving system.[1]

[1] `http://coq.inria.fr/`

The use of explicit proof terms to certify the safety of mobile code goes back to Necula and Lee [13, 12]. They used the LF logical framework with some optimizations for proof representation. Proof-carrying authorization was proposed by Appel and Felten [1] and then realized for the first time by Bauer, Schneider, and Felten [5, 3]. It is now one of the cornerstones of the Grey project [4] for universal distributed access control at Carnegie Mellon University.

The technique of representing proofs of meta-theorems by relations is my own work [14, 15], eventually leading to the Twelf system [17] which has many contributors. Type reconstruction for Twelf was already sketched in the early work [15], followed the first foray into mode checking and termination checking [19] later extended by Pientka [18]. Totality checking [21] was a further development of meta-proofs that were correct by construction proposed by Schürmann [20]. The example in this lecture can be easily verified in Twelf. For more on Twelf see the Twelf home page.[2]

Relational representation of meta-theory has recently been shown to be sufficiently powerful to mechanize the theory of full-scale programming languages (Typed Assembly Language [7] and Standard ML [11]). For further references and a more general introduction to logical frameworks, see [16].

18.10 Exercises

Exercise 18.1 *We can translate the conditions on polymorphism, namely that that term constructors be type-preserving and predicates be parametric, to conditions on dependency. Give such a parallel definition.*

Further, give examples that show the need for type information to be carried during unification to avoid creating ill-typed terms if these conditions are violated.

Finally, discuss which of these, if any, would be acceptable in the case of dependent types.

Exercise 18.2 *Revisit the proof that the empty list is the right unit for* append *(Exercise 3.5 from Assignment 2) and represent it formally as a relation between deductions.*

Exercise 18.3 *Revisit the proof that addition is commutative (Exercise 3.4 from Assignment 2) and represent it formally as a relation between deductions.*

Exercise 18.4 *Revisit the proof that append is associative (Exercise 3.6 from Assignment 2) and represent it formally as a relation between deductions.*

[2] `http://www.twelf.org/`

Exercise 18.5 *If we do not want to change the logic programming engine to produce proof objects, we can transform a program globally by adding an additional argument to every predicate, denoting a proof term.*

Formally define this transformation and prove its correctness.

18.11 References

[1] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[2] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[3] Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.

[4] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, Singapore, September 2005. Springer Verlag LNCS 3650.

[5] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.

[6] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[7] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. *ACM Transactions on Computational Logic*, 2006. To appear.

[8] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[10] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[11] Daniel K. Lee, Karl Crary, and Robert Harper. Mechanizing the metatheory of Standard ML. Technical Report CMU-CS-06-138, Carnegie Mellon University, 2006.

[12] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[13] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, October 1996.

[14] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[15] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[16] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[17] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[18] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. In Carsten Schürmann, editor, *Workshop on Automation of Proofs by Mathematical Induction*, Pittsburgh, Pennsylvania, June 2000.

[19] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, edi-

tor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[20] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[21] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

15-819K: Logic Programming

Lecture 19

Verifying Progress

Frank Pfenning

November 2, 2006

In this lecture we discuss another logic program analysis, namely verifying the progress property. Progress guarantees that a predicate can never fail for arbitrary values of its input arguments. Together with termination this guarantees that a predicate is total in its given input arguments. As sketched in the previous lecture, this an important piece in the general technique of verifying properties of logic programs by reasoning about proof terms.

19.1 The Progress Property

Progress in general just says that during the execution of a program we have either finished computation with a value, or we can make a further step. In particular, computation can never "get stuck". In logic programming this translates to saying the computation can never fail. This requires an understanding of the intended input and output arguments of a predicate, as well as the domain on which it is to be applied.

Returning to the well-worn example of addition,

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

the plus predicate is total in the first two arguments, assuming they are natural numbers. It is not total in the first and third argument, because a query such as $\mathsf{plus}(\mathsf{s}(\mathsf{z}), N, \mathsf{z})$ will fail. Totality decomposes into two subquestions, namely progress and termination, since we always assume the program is well-typed and well-moded. Termination is easy to see here because the

first argument decreases strictly in each recursive call. Progress is also easy to see because the first argument must be a natural number, and therefore be either of the form z or $\mathsf{s}(m)$ for some m, and the second argument can be anything because both clause heads have a variable in that position.

Even though the principal application of progress is probably the verification of proof of metatheorems presented in relational form, progress can also be used to check that some given predicates are total functions (although ruling out multi-valued functions requires another step). This may provide the programmer with additional confidence that no cases in the definition of a logic program were missed.

19.2 The Right Semantic Starting Point

As repeatedly emphasized, finding the right semantic starting point for an analysis is the key to obtaining a simple, predictable system and the easiest proof of correctness. For progress, the residuated form of the program is somewhat difficult to deal with. Consider the simple form of $\mathsf{plus}(+,+,-)$ above (easily seen to satisfy progress) and the residuated form

$$\begin{aligned}\mathsf{plus}(x_1, x_2, x_3) \leftarrow\ & (\exists N.\, x_1 \doteq \mathsf{z} \wedge x_2 \doteq N \wedge x_2 \doteq N) \\ \vee\ & (\exists M.\, \exists N.\, \exists P.\, x_1 \doteq \mathsf{s}(M) \wedge x_2 \doteq N \wedge x_3 \doteq \mathsf{s}(P) \wedge \mathsf{plus}(M, N, P))\end{aligned}$$

on which it is more difficult to discern the same property. Moreover, failure plays no role in the progress property because, in fact, it is never permitted to occur, so the semantics should not need to carry a failure continuation.

Hence we return to a fairly early semantics, in which the subgoal stack is explicit, but not the failure continuation. On the other hand, the substitution for the variables is crucial, so we make that explicit. Recall that there is a fixed program Γ with a set of closed clauses.

$$\frac{\tau \vdash G_1 \mathbin{/} G_2 \wedge S}{\tau \vdash G_1 \wedge G_2 \mathbin{/} S} \qquad \frac{\tau \vdash G_2 \mathbin{/} S}{\tau \vdash \top \mathbin{/} G_2 \wedge S} \qquad \frac{}{\tau \vdash \top \mathbin{/} \top}$$

$$\frac{(\forall \mathbf{x}.\, P' \leftarrow G) \in \Gamma \quad P'\rho \doteq P\tau \mid \theta \quad \tau\theta, \rho\theta \vdash G \mathbin{/} S}{\tau \vdash P \mathbin{/} S}$$

In the last rule, ρ is a substitution renaming $\mathbf{x}$ to a new set of logic variables $\mathbf{X}$, that is, $\mathrm{dom}(\rho) = \mathbf{x}$, $\mathrm{cod}(\rho) \cap \mathrm{cod}(\tau) = \emptyset$. We also assume that the variables $\mathbf{x}$ have been renamed so that $\mathbf{x} \cap \mathrm{dom}(\tau) = \emptyset$.

From this semantics it is easily seen that progress is a question regarding atomic goals, because the cases for conjunction and truth always apply.

19.3 Input and Output Coverage

Focusing in, we rewrite the rules for predicate calls assuming the predicate $p(\mathbf{t}, \mathbf{s})$ has a mode declaration which divides the arguments into input arguments $\mathbf{t}$, which come first, and output arguments $\mathbf{s}$, which come second.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G) \in \Gamma \quad (\mathbf{t}'\rho, \mathbf{s}'\rho) \doteq (\mathbf{t}\tau, \mathbf{s}\tau) \mid \theta \quad \tau\theta, \rho\theta \vdash G \;/\; S}{\tau \vdash p(\mathbf{t}, \mathbf{s}) \;/\; S}$$

We have ensured progress if such a clause and unifier θ always exist.

Breaking it down a bit further, we see we must have

> *There exists a θ such that (1) $\mathbf{t}'\rho\theta = \mathbf{t}\tau\theta$, and (2) $\mathbf{s}'\rho\theta = \mathbf{s}\tau\theta$ where $\mathbf{t}'$ are the input terms in the clause head, $\mathbf{s}'$ are the output terms in the clause head, $\mathbf{t}$ are the input arguments to the predicate call, and $\mathbf{s}$ are the output arguments in the predicate call.*

We refer to part (1) as *input coverage* and part (2) as *output coverage*. In the problem analysis above a single substitution θ is required, but we will approximate this by two separate checks. In the next two sections we will describe the analysis for the two parts of coverage checking. We preview them here briefly.

For input coverage we need to recall the assumption that predicates are well-moded. This means that the input arguments in the call, $\mathbf{t}\tau\theta$ will be ground. Hence input coverage is satisfied if for any sequence $\mathbf{t}$ of ground terms of the right types, there exists a clause head such that its input arguments $\mathbf{t}'$ can be instantiated to $\mathbf{t}$.

Output coverage is trickier. The problem is that mode analysis does not tell us anything about the output argument $\mathbf{s}\tau$ of the call $p(\mathbf{t}\tau, \mathbf{s}\tau)$. What we know if that if p succeeds with substitution $\tau\theta'$, then $\mathbf{s}\tau\theta'$ will be ground, but this does not help. From examples, like plus above, we can observe that output coverage is satisfied because the output argument of the call (in the second clause for plus it is P) is a variable, and will remain a variable until the call is made. This means we have to sharpen mode checking to verify that some variables remain free, which we tackle below.

19.4 Input Coverage

Given a program Γ and a predicate $p : (\boldsymbol{\tau}, \boldsymbol{\sigma}) \rightarrow o$ with input arguments of type $\boldsymbol{\tau}$. We say that p satisfies input coverage in Γ if for any sequence of ground terms $\mathbf{t} : \boldsymbol{\tau}$ there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G$ and a substitution $\theta : (\mathbf{x}{:}\boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}$.

For the description of the algorithm, we will need a slightly more general form. We write $\Delta \vdash \Gamma_p \gg \mathbf{t}$ (read: Γ_p immediately covers $\mathbf{t}$) if there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G$ in Γ_p and a substitution $\Delta \vdash \theta : (\mathbf{x} : \boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}$. We write $\Delta \vdash \Gamma_p > \mathbf{t}$ (read: Γ_p covers $\mathbf{t}$) if for every ground instance $\mathbf{t}\sigma$ with $\sigma : \Delta$ there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}) \leftarrow G$ in Γ_p and a subsitution $\Delta \vdash \theta : (\mathbf{x}{:}\boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}\sigma$. Clearly, immediate coverage implies coverage, but not vice versa.

We reconsider the plus predicate, with the first two arguments considered as inputs.

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

By the preceding remark, in order to show that input coverage holds, it is sufficient to show that

$$x_1{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (x_1, x_2).$$

Clearly, immediate coverage does not hold, because x_1 is not an instance of either z or $\mathsf{s}(M)$. On the other hand, x_2 is an instance of N.

At this point we need to exploit the assumption $x_1{:}\mathsf{nat}$ by applying an appropriate left rule. This is acceptable because we move from the usual open world assumption (any predicate and type is inherently open-ended) to the closed world assumption (all predicates and types are given completely by their definition). The closed world assumption is necessary because progress (and coverage) can only be establish with respect to a fixed set of types and clauses and could immediately be violated by new declarations (e.g., the additional declaration $\omega : \mathsf{nat}$ causes input coverage for plus to fail).

To see what the left rules would look like, we can take a short detour through type predicates. The declarations

$$\begin{array}{lcl}\mathsf{z} & : & \mathsf{nat}.\\ \mathsf{s} & : & \mathsf{nat} \rightarrow \mathsf{nat}.\end{array}$$

correspond to

$$\begin{array}{l}\mathsf{nat}(\mathsf{z}).\\ \mathsf{nat}(\mathsf{s}(N)) \leftarrow \mathsf{nat}(N).\end{array}$$

The iff-completion yields

$$\mathsf{nat}(N) \quad \leftrightarrow \quad N \doteq \mathsf{z} \vee \exists N'.\, N \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N').$$

The left rule for $\mathsf{nat}(x)$ (which is not the most general case, but sufficient for our purposes) for an arbitrary judgment J on the right-hand side can then be derived as

$$\dfrac{\dfrac{\vdash J(\mathsf{z}/x)}{x \doteq \mathsf{z} \vdash J} \quad \dfrac{\dfrac{\mathsf{nat}(x') \vdash J(\mathsf{s}(x')/x)}{x \doteq s(x') \wedge \mathsf{nat}(x') \vdash J}}{\exists N'.\, x \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N') \vdash J}}{x \doteq \mathsf{z} \vee \exists N'.\, x \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N') \vdash J}}{\mathsf{nat}(x) \vdash J}$$

or, in summary:

$$\dfrac{\vdash J(\mathsf{z}/x) \quad \mathsf{nat}(x') \vdash J(\mathsf{s}(x')/x)}{\mathsf{nat}(x) \vdash J}$$

Translated back to types:

$$\dfrac{\Delta \vdash J(\mathsf{z}/x) \quad \Delta, x'{:}\mathsf{nat} \vdash J(\mathsf{s}(x')/x)}{\Delta, x{:}\mathsf{nat} \vdash J}$$

Using this rule, we can now prove our goal:

$$\dfrac{x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (\mathsf{z}, x_2) \quad x_1'{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (\mathsf{s}(x_1'), x_2)}{x_1{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (x_1, x_2)}$$

Both of the premisses now follow by immediate coverage, using the first clause for the first premiss and the second clause for the second premiss, using the critical rule

$$\dfrac{\Delta \vdash \Gamma_p \gg \mathbf{t}}{\Delta \vdash \Gamma_p > \mathbf{t}}$$

For immediate coverage, there is but one rule.

$$\dfrac{(\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G) \in \Gamma_p \quad \mathbf{t}'\theta = \mathbf{t} \quad \text{for } \Delta \vdash \theta : (\mathbf{x}{:}\boldsymbol{\tau}')}{\Delta \vdash \Gamma_p \gg \mathbf{t}}$$

We do not write out the left rules, but it should be clear how to derive them from the type declarations, at least for simple types. We call this process *splitting* of a variable $x{:}\tau$.

An interesting aspect of the left rules is that they are asynchronous. However, always applying them leads to non-termination, so we have to follow some terminating strategy. This strategy can be summarized informally as follows, given a goal $\Delta \vdash \Gamma_p > \mathbf{t}$.

1. Check if $\Delta \vdash \Gamma_p \gg \mathbf{t}$. If so, succeed.
2. If not, pick a variable $x{:}\tau$ in Δ and apply inversion as sketched above. This yields a collection of subgoals $\Delta_i \vdash \Gamma_p \gg \mathbf{t}_i$. Solve each subgoal.

Picking the right variable to split is crucial for termination. Briefly, we pick a variable x that was involved in a clash $f(\mathbf{t}'') \doteq x$ when attempting immediate coverage, where $f(\mathbf{t}'')$ is a subterm of $\mathbf{t}'$. Now one of the possibilities for x will have f as it top-level function symbol, reducing the clash the next time around. Thus the splitting process is bounded by the total size of the input terms in Γ_p. See the reference below for further discussion and proof of this fact.

19.5 Output Coverage

Output coverage is to ensure that for every goal $p(\mathbf{t}, \mathbf{s})$ encountered while executing a program, the output positions $\mathbf{s}'$ of the relevant clause head $p(\mathbf{t}', \mathbf{s}')$ are an instance of $\mathbf{s}$ (if $\mathbf{t}$ and $\mathbf{t}'$ unify). The problem is that ordinary mode checking does not tell us anything about $\mathbf{s}$: we do not know whether it will be ground or partially ground or consist of all free variables. However, if we knew that $\mathbf{s}$ consisted of pairwise distinct free variables when the goal $p(\mathbf{t}, \mathbf{s})$ arose, then output coverage would be satisfied since the variables in $\mathbf{s}$ cannot occur in a clause head and therefore the unification process must succeed.

So we can guarantee output coverage with a sharpened mode-checking process where an output arguments must be distinct free variables when a predicate is invoked. Moreover, they must become ground by the time the predicate suceeds. This is actually very easy: just change the abstract domain of the mode analysis from $\mathsf{u} > \mathsf{g}$ (unknown and ground) to $\mathsf{f} > \mathsf{g}$ (free and ground). If we also have bidirectional arguments in addition to input and output we three abstract values with $\mathsf{f} > \mathsf{u} > \mathsf{g}$. The remainder of the development is just as in a previous lecture (see Exercise 19.1). The only slightly tricky aspect is that the output arguments must be *distinct* free variables, otherwise the individual substitutions may not compose to one for all output arguments simultaneously.

Returning to our example,

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

the only output argument is P, which is indeed a free variable when that subgoal is executed.

We show a couple of cases for failure of output coverage, to illustrate some points. Assume we have already checked progress for plus. The program

$$\mathsf{test} \leftarrow \mathsf{plus}(\mathsf{z}, \mathsf{z}, \mathsf{s}(P)).$$

trivially satisfies input coverage, but yet fails (and hence cannot satisfy progress). This is because the output argument in the only call is $\mathsf{s}(P)$, which is not a free variable. This will be noted by the sharpened mode checker.

Similarly, the program

$$\begin{array}{l}\mathsf{test} \leftarrow \\ \quad \mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P), \\ \quad \mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), P).\end{array}$$

trivially satisfies input coverage but it does not pass the sharpened mode checker because the second occurrence of P will be ground (from the first call) rather than free when the second subgoal is executed. And, indeed, plus will fail and hence cannot satisfy progress.

Finally, a the predicate $\mathsf{nexttwo}(+, -, -)$

$$\mathsf{nexttwo}(N, \mathsf{s}(N), \mathsf{s}(\mathsf{s}(N))).$$

satisfies progress, but

$$\mathsf{test} \leftarrow \mathsf{nexttwo}(\mathsf{s}(\mathsf{z}), P, P).$$

does not, because the two occurrences of P would have to be $\mathsf{s}(\mathsf{s}(\mathsf{z}))$ and $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$ simultaneously.

19.6 Historical Notes

Progress and coverage do not appear to have received much attention in the logic programming literature, possibly because they requires types to be interesting, and their main application lies in verifying proofs of meta-theorems which is a recent development. An algorithm for coverage in the richer setting with dependent types is given by Schürmann and myself [1], which also contains some pointers to earlier literature in functional programming.

19.7 Exercises

Exercise 19.1 *Write out the rules for a sharpened mode checker with only input and output arguments where output arguments must be distinct free variables when a predicate is invoked.*

19.8 References

[1] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

15-819K: Logic Programming

Lecture 20

Bottom-Up Logic Programming

Frank Pfenning

November 7, 2006

In this lecture we return to the view that a logic program is defined by a collection of inference rules for atomic propositions. But we now base the operational semantics on reasoning forward from facts, which are initially given as rules with no premisses. Every rule application potentially adds new facts. Whenever no more new facts can be generated we say forward reasoning *saturates* and we can answer questions about truth by examining the saturated database of facts. We illustrate bottom-up logic programming with several programs, including graph reachability, CKY parsing, and liveness analysis.

20.1 Bottom-Up Inference

We now return the very origins of logic programming as an operational interpretation of inference rules defining atomic predicates. As a reminder, consider the definition of even.

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evss}$$

This works very well on queries such as $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))$ (which succeeds) and $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))$ (which fails). In fact, the operational reading of this program under goal-directed search constitutes a decision procedure for ground queries $\mathsf{even}(n)$.

This specification makes little sense under an alternative interpretation where we eagerly apply the inference rules in the forward direction, from the premisses to the conclusion, until no new facts can be deduced. The

problem is that we start with $\mathsf{even}(\mathsf{z})$, then obtain $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$, and so on, but we never terminate.

It would be too early to give up on forward reasoning at this point. As we have seen many times, even in backward reasoning a natural specification of a predicate does not necessarily lead to a reasonable implementation. We can implement a test whether a number is even via reasoning by contradication. We seed our database with the claim that n is not even and derive consequences from that assumption. If we derive a contradictory fact we know thate $\mathsf{even}(n)$ must be true. If not (and our rules are complete), then $\mathsf{even}(n)$ must be false. We write $\mathsf{odd}(n)$ for the proposition that n is not even. Then we obtain the following specification

$$\frac{\mathsf{odd}(\mathsf{s}(\mathsf{s}(N)))}{\mathsf{odd}(N)}$$

to be used for forward reasoning. This single rule obviously saturates because the argument to odd becomes smaller in every rule application.

What is not formally represented in this program is how we initialize our database (we assume $\mathsf{odd}(n)$), and how we interpret the saturated database (we check if $\mathsf{odd}(\mathsf{z})$ was deduced). In a later lecture we will see that it is possible to combine forward and backward reasoning to makes those aspects of an algorithm also part of its implementation.

The strategy of this example, proof by contradiction, does not always work, but there are many cases where it does. One should check if the predicate is decidable as a first test. We will see further examples later, specifically the treatment of unification in the next lecture.

20.2 Graph Reachability

Assuming we have a specification of $\mathsf{edge}(x, y)$ whenever there is an edge from node x to node y, we can specify reachability $\mathsf{path}(x, y)$ with the rules

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)} \qquad \frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

During bottom-up inference these rules will saturate when they have constructed the transitive closure of the edge relation. During backward reasoning these rules may not terminate (if there are cycles), or be very inefficient (if there are many paths compared to the number of nodes).

In the forward direction the rules will always saturate. We can also give, just from the rules, a complexity analysis of the saturation algorithm.

20.3 Complexity Analysis

McAllester [3] proved a so-called meta-complexity result which allows us to analyze the structure of a bottom-up logic program and obtain a bound for its asymptotic complexity. We do not review the result or its proof in full detail here, but we sketch it so it can be applied to several of the programs we consider here. Briefly, the result states that the complexity of a bottom-up logic program is $O(|R(D)| + |P_R(R(D))|)$, where $R(D)$ is the saturated database (writing here D for the initial database) and $P_R(R(D))$ is the set of prefix firings of rules R in the saturated database.

The number *prefix firings* for a given rule is computed by analyzing the premisses of the rule from left to right, counting in how many ways it could match facts in the saturated database. Matching an earlier premiss will fix its variables, which restricts the number of possible matches for later premisses.

For example, in the case of the transitive closure program, assume we have e edges and n vertices. Then in the completed database there can be at most n^2 facts $\mathsf{path}(x, y)$, while there are always exactly e facts $\mathsf{edge}(x, y)$. The first rule

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)}$$

can therefore always match in e ways in the completed database. We analyze the premisses of the second rule

$$\frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

from left to right. First, $\mathsf{edge}(X, Y)$ can match the database in $O(e)$ ways, as before. This match fixes Y, so there are now $O(n)$ ways that the second premiss could match a fact in the saturated database (each vertex is a candidate for Z). This yields $O(e \cdot n)$ possible prefix firings.

The size of the saturated database is $O(e+n^2)$, and the number of prefix firings of the two rules is $O(e + e \cdot n)$. Therefore the overall complexity is $O(e \cdot n + n^2)$. Since there are up to n^2 edges in the graph, we get a less informative bound of $O(n^3)$ expressed entirely in the number of vertices n.

20.4 CKY Parsing

Another excellent example for bottom-up logic programming and complexity analysis is a CKY parsing algorithm. This algorithm assumes that

the grammar is in Chomsky-normal form, where productions all have the form

$$\begin{array}{lcl} x & \Rightarrow & y\,z \\ x & \Rightarrow & a \end{array}$$

where x, y, and z stand for non-terminals and a for terminal symbols. The idea of the algorithm is to use the grammar production rules from right to left to compute which sections of the input string can be parsed as which non-terminals.

We initialize the database with facts $\mathsf{rule}(x, \mathsf{char}(a))$ for every grammar production $x \Rightarrow a$ and $\mathsf{rule}(x, \mathsf{jux}(y, z))$ for every production $x \Rightarrow y\,z$. We further represent the input string $a_1 \ldots a_n$ by assumptions $\mathsf{string}(i, a_i)$. For simplicity, we represent numbers in unary form.

Our rules will infer propositions $\mathsf{parse}(x, i, j)$ which we will deduce if the substring $a_i \ldots a_j$ can be parsed as an x. Then the program is represented by the following two rules, to be read in the forward direction:

$$\frac{\mathsf{rule}(X, \mathsf{char}(A)) \quad \mathsf{string}(I, A)}{\mathsf{parse}(X, I, I)} \qquad \frac{\mathsf{rule}(X, \mathsf{jux}(Y, Z)) \quad \mathsf{parse}(Y, I, J) \quad \mathsf{parse}(Z, s(J), K)}{\mathsf{parse}(X, I, K)}$$

After saturating the database with these rules we can see if the whole string is in the language generated by the start symbol s by checking if the fact $\mathsf{parse}(s, \mathsf{s}(\mathsf{z}), n)$ is in the database.

Let g be the number of grammar productions and n the length of the input string. In the completed database we have g grammar rules, n facts $\mathsf{string}(i, a)$, and at most $O(g \cdot n^2)$ facts $\mathsf{parse}(x, i, j)$.

Moving on to the rules, in the first rule there are $O(g)$ ways to match the grammar rule (which fixes A) and then n ways to match $\mathsf{string}(I, A)$, so we have $O(g \cdot n)$. The second rule, again we have $O(g)$ ways to match the grammar rule (which fixes X, Y, and Z) and then $O(n^2)$ ways to match $\mathsf{parse}(Y, I, J)$. In the third premiss now only K is unknown, giving us $O(n)$ way to match it, which means $O(g \cdot n^3)$ prefix firings for the second rule.

These considerations give us an overall complexity of $O(g \cdot n^3)$, which is also the traditional complexity bound for CKY parsing.

20.5 Liveness Analysis

We consider an application of bottom-up logic programming in program analysis. In this example we analyze code in a compiler's intermediate

language to find out which variables are live or dead at various points in the program. We say a variable is *live* at a given program point l if its value will be read before it is written when computation reaches l. This information can be used for optimization and register allocation.

Every command in the language is labeled by an address, which we assume to be a natural number. We use l and k for labels and w, x, y, and z for variables, and op for binary operators. In this stripped-down language we have the following kind of instructions. A representation of the instruction as a logical term is given on the right, although we will continue to use the concrete syntax to make the rules easier to read.

l	:	$x = op(y, z)$	$\mathsf{inst}(l, \mathsf{assign}(x, op, y, z))$
l	:	if x goto k	$\mathsf{inst}(l, \mathsf{if}(x, k))$
l	:	goto k	$\mathsf{inst}(l, \mathsf{goto}(k))$
l	:	halt	$\mathsf{inst}(l, \mathsf{halt})$

We use the proposition $x \neq y$ to check if two variables are distinct and write $\mathsf{s}(l)$ for the successor location to l which contains the next instruction to be executed unless the usual control flow is interrupted.

We write $\mathsf{live}(w, l)$ if we have inferred that variable w is live at l. This is an over-approximation in the sense that $\mathsf{live}(w, l)$ indicates that the variable *may* be live at l, although it is not guaranteed to be read before it is written. This means that any variable that is not live at a given program point is definitely *dead*, which is the information we want to exploit for optimization and register allocation.

We begin with the rules for assignment $x = op(y, z)$. The first two rules just note the use of variables as arguments to an operator. The third one propagates liveness information backwards through the assignment operator. This is sound for any variable, but we would like to achieve that x is not seen as live before the instruction $x = op(y, z)$, so we verify that $W \neq X$.

$$\frac{L : X = Op(Y, Z)}{\mathsf{live}(Y, L)} \qquad \frac{L : X = Op(Y, Z)}{\mathsf{live}(Z, L)} \qquad \frac{\begin{array}{l} L : X = Op(Y, Z) \\ \mathsf{live}(W, \mathsf{s}(L)) \\ W \neq X \end{array}}{\mathsf{live}(W, L)}$$

The rules for jumps propagate liveness information backwards. For unconditional jumps we look at the target; for conditional jumps we look both at the target and the next statement, since we don't analyze whether the

condition may be true or false.

$$\frac{L : \mathsf{goto}\ K \quad \mathsf{live}(W, K)}{\mathsf{live}(W, L)} \qquad \frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K \quad \mathsf{live}(W, K)}{\mathsf{live}(W, L)} \qquad \frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K \quad \mathsf{live}(W, \mathsf{s}(L))}{\mathsf{live}(W, L)}$$

Finally, the variable tested in a conditional is live.

$$\frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K}{\mathsf{live}(X, L)}$$

For the complexity analysis, let n be the number of instructions in the program and v be the number of variables. The size of the saturated database is $O(v \cdot n)$, since all derived facts have the form $\mathsf{live}(X, L)$ where X is a variable and L is the label of an instruction. The prefix firings of all 7 rules are similarly bounded by $O(v \cdot n)$: there are n ways to match the first instruction and then at most v ways to match the second premiss (if any). Hence the overall complexity is bounded by $O(v \cdot n)$.

20.6 Functional Evaluation

As a last example in this lecture we present an algorithm for functional call-by-value evaluation. Our language is defined by

$$e ::= x \mid \lambda x.\, e \mid e_1\, e_2$$

We assume that substitution on terms is a primitive so we can avoid implementing it explicitly (see Exercise 20.4). Such an assumption is not unreasonable. For example, in the LolliMon language which provides both top-down and bottom-up logic programming, substitution is indeed built-in. Since we are only interested in evaluating closed terms, all values here have the form $\lambda x.\, e$.

We use three predicates:

$\mathsf{eval}(e)$	evaluate e
$e \rightarrow^* e'$	e reduces to e'
$e \hookrightarrow v$	e evaluates to v.

We seed the database with $\mathsf{eval}(e)$, saturate it, and then read off the value as $e \hookrightarrow v$. Of course, since this is the untyped λ-calculus, saturation is not guaranteed.

The first rules propagate the information about which terms are to be evaluated.

$$\frac{\mathsf{eval}(\lambda x.\, e)}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{\mathsf{eval}(e_1\, e_2)}{\mathsf{eval}(e_1)} \quad \frac{\mathsf{eval}(e_1\, e_2)}{\mathsf{eval}(e_2)}$$

In case we had an application we have to gather the results, substitute the argument into the body of the function, and recursively evaluate the result. This generates a reduction from which we need to initiate evaluation. Finally, we need to compose reductions to obtain the final value.

$$\frac{\begin{array}{l}\mathsf{eval}(e_1\, e_2)\\ e_1 \hookrightarrow \lambda x.\, e_1'\\ e_2 \hookrightarrow v_2\end{array}}{e_1\, e_2 \longrightarrow^* e_1'(v_2/x)} \qquad \frac{e \longrightarrow^* e'}{\mathsf{eval}(e')} \qquad \frac{e \longrightarrow^* e' \quad e' \hookrightarrow v}{e \hookrightarrow v}$$

As an example, consider the following database saturation process.

$$\begin{array}{l}\mathsf{eval}((\lambda x.\, x)\, (\lambda y.\, y))\\ \mathsf{eval}(\lambda x.\, x)\\ \mathsf{eval}(\lambda y.\, y)\\ \lambda x.\, x \hookrightarrow \lambda x.\, x\\ \lambda y.\, y \hookrightarrow \lambda y.\, y\\ (\lambda x.\, x)\, (\lambda y.\, y) \longrightarrow^* \lambda y.\, y\\ (\lambda x.\, x)\, (\lambda y.\, y) \hookrightarrow \lambda y.\, y\end{array}$$

This form of evaluation may seem a bit odd, compared to the usual top-down formulation (again, assuming substitution as a primitive)

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e_1' \quad e_2 \hookrightarrow v_2 \quad e_1'(v_2/x) \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

However, it does have some advantages. If we proved its completeness (see Exercise 20.5), we would get some theorems for free. For example, it is easy to see that $\mathsf{eval}((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))$ saturates without producing a value for the application:

$$\begin{array}{l}\mathsf{eval}((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))\\ \mathsf{eval}(\lambda x.\, x\, x)\\ (\lambda x.\, x\, x) \hookrightarrow (\lambda x.\, x\, x)\\ (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \longrightarrow^* (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)\end{array}$$

At this point the database is saturated. This proves that the evaluation of $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ fails. In the top-down semantics (that is, with backward chaining as in Prolog), such a query would fail to terminate instead unless we added some kind of loop detection.

Note that the bottom-up program for evaluation, which consists of six rules, cannot be analyzed with McAllester's technique, because in the conclusion of the rule for reduction a new term $e_1'(v_2/x)$ is created. We can therefore not bound the size of the completed database. And, in fact, the saturation may fail to terminate (see Exercise 20.7).

20.7 Variable Restrictions

Bottom-up logic programming, as considered by McAllester, requires that every variable in the conclusion of a rule also appears in a premiss. This means that every generated fact will be ground. This is important for saturation and complexity analysis because a fact with a free variable could stand for infinitely many instances.

Nonetheless, bottom-up logic programming can be generalized in the presence of free variables and we will do this in a later lecture.

20.8 Historical Notes

The bottom-up interpretation of logic programs goes back to the early days of logic programming. See, for example, the paper by Naughton and Ramakrishnan [4].

There are at least three areas were logic programming specification with a bottom-up semantics has found significant applications: deductive databases, decision procedures, and program analysis. Unification, as present in the next lecture, is an example of a decision procedure for unifiability. Liveness analysis is an example of program analysis due to McAllester [3], who was particularly interested in describing program analysis algorithms at a high level of abstraction so their complexity would be self-evident. This was later refined by Ganzinger and McAllester [1, 2] by allowing deletions in the database. We treat this in a later lecture where we generalize bottom-up inference to linear logic.

20.9 Exercises

Exercise 20.1 *Write a bottom-up logic program for addition (`plus/3`) on numbers in unary form and then extend it to multiplication (`times/3`).*

Exercise 20.2 *Consider the following variant of graph reachability.*

$$\frac{\mathsf{edge}(X,Y)}{\mathsf{path}(X,Y)} \qquad \frac{\mathsf{path}(X,Y) \quad \mathsf{path}(Y,Z)}{\mathsf{path}(X,Z)}$$

Perform a McAllester-style complexity analysis and compare the infered complexity with the one given in lecture.

Exercise 20.3 *The set of prefix firings depends on the order of the premisses. Give an example to demonstrate this.*

Exercise 20.4 *Extend the bottom-up evaluation semantics for λ-terms by adding rules to compute the substitutions $e(v/x)$. You may assume that v is closed, and that the necessary tests on variable names can be performed.*

Exercise 20.5 *Relate the bottom-up and top-down version of evaluation of λ-terms to each other by an appropriate pair of theorems.*

Exercise 20.6 *Add pairs to the evaluation semantics, together with first and second projections. A pair should only be a value if both components are values, that is, pairs are eagerly evaluated.*

Exercise 20.7 *Give an example which shows that saturation of evaluation for λ-terms may fail to terminate.*

20.10 References

[1] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T.Nipkow R.Goré, A.Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

[2] Harald Ganzinger and David A. McAllester. Logical algorithms. In P. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[3] Dave McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[4] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson*, pages 640–700. MIT Press, Cambridge, Massachusetts, 1991.

15-819K: Logic Programming

Lecture 21

Forward Chaining

Frank Pfenning

November 9, 2006

In this lecture we go from the view of logic programming as derived from inference rules for atomic propositions to one with explicit logical connectives. We have made this step before in order to describe the backward-chaining semantics of top-down logic programming as in Prolog. Here we instead describe the forward-chaining semantics of bottom-up logic programming. We use this to prove the correctness of an earlier example, and introduce an algorithm for unification as another example.

21.1 The Database as Context

When we try to formalize the semantics of bottom-up logic programming and saturation, one of the first questions to answer is how to represent the database from a logical point of view. Perhaps surprisingly at first, it ends up as a context of *assumptions*. We interpret a rule

$$\frac{P_1\ \mathit{true} \ldots P_n\ \mathit{true}}{P\ \mathit{true}}$$

as license to add the assumption $P\ \mathit{true}$ if we already have assumption $P_1\ \mathit{true}$ through $P_n\ \mathit{true}$. This means we actually have to turn the rule upside down to obtain the left rule

$$\frac{\Gamma, P_1\ \mathit{true}, \ldots, P_n\ \mathit{true}, P\ \mathit{true} \vdash C\ \mathit{true}}{\Gamma, P_1\ \mathit{true}, \ldots, P_n\ \mathit{true} \vdash C\ \mathit{true}}$$

Since in the case of (non-linear) intuitionistic logic, the context permits weakening and contraction, this step only represents progress if P is not

already in Γ or among the P_i. We therefore stop forward chaining if none of the inferences would make progress and say the database, represented as the context of assumptions, is saturated.

For now we will view logical deduction as ground deduction and return to the treatment of free variables in the next lecture. However, the inference rules to infer ground facts may still contain free variables. If we reify inference rules as logical implications and collect them in a fixed context Γ_0 we obtain the next version of the above rule (omitting '$true$'):

$$\frac{\begin{array}{ll}(\forall \mathbf{x}.\, P'_1 \wedge \ldots \wedge P'_n \supset P') \in \Gamma_0 & \mathrm{dom}(\theta) = \mathbf{x} \\ P'_i\theta = P_i \quad \text{for all } 1 \leq i \leq n & \mathrm{cod}(\theta) = \emptyset \qquad \Gamma, P_1, \ldots, P_n, P'\theta \vdash C\end{array}}{\Gamma, P_1, \ldots, P_n \vdash C}$$

To model saturation, we would restrict the rule to the case where $P'\theta \notin \Gamma, P_1, \ldots, P_n$. Moreover, the set of free variables in P' should be a subset of the variables in $P_1, \ldots, P_n$ so that $P'\theta$ is ground without having to guess a substitution term.

Note that the right-hand side C remains unchanged and unreferenced in the process of forward chaining. Later, when we are interested in combining forward and backward chaining we will have to pay some attention to the right-hand side. For now we leave the processes of creating the initial database and reading off an answer from the saturated database informal and concentrate on the forward chaining itself.

21.2 Even and Odd, Revisited

We revisit the two programs for checking if an given number is even in order to see how we can reason about the correctness of forward chaining programs. Recall first the definition of even, which has a natural backward chaining interpretation.

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{evz} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evss}$$

Next, the rule for odd, which is our notation of the property of not being even.

$$\frac{\mathsf{odd}(\mathsf{s}(\mathsf{s}(N)))}{\mathsf{odd}(N)}$$

To see if $\mathsf{even}(n)$ we seed the database with $\mathsf{odd}(n)$, forward chain to saturation and then check if we have derived a contradiction, namely $\mathsf{odd}(\mathsf{z})$.

The correctness of the forward chaining program can be formulated as:

$\mathsf{even}(n)$ *true iff* $\mathsf{odd}(n)$ *true* $\vdash \mathsf{odd}(\mathsf{z})$ *true*

The intent is that we only use forward chaining rules for the second judgment, that is, we work entirely on the left except for an initial sequent to close off the derivation.

Theorem 21.1 *If* $\mathsf{even}(n)$ *true then* $\mathsf{odd}(n)$ *true* $\vdash \mathsf{odd}(\mathsf{z})$ *true.*

Proof: By induction on the deduction $\mathcal{D}$ of $\mathsf{even}(n)$ *true*

Case: $\mathcal{D} = \dfrac{}{\mathsf{even}(\mathsf{z})}$ where $n = \mathsf{z}$.

$\mathsf{odd}(\mathsf{z}) \vdash \mathsf{odd}(\mathsf{z})$ By hypothesis rule

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(n')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(n')))}$ where $n = \mathsf{s}(\mathsf{s}(n'))$.

$\mathsf{odd}(n') \vdash \mathsf{odd}(\mathsf{z})$ By ind. hyp. on $\mathcal{D}'$
$\mathsf{odd}(\mathsf{s}(\mathsf{s}(n'))), \mathsf{odd}(n') \vdash \mathsf{odd}(\mathsf{z})$ By weakening
$\mathsf{odd}(\mathsf{s}(\mathsf{s}(n'))) \vdash \mathsf{odd}(\mathsf{z})$ By forward chaining

□

The last step in the second case of this proof is critical. We are trying to show that $\mathsf{odd}(\mathsf{s}(\mathsf{s}(n'))) \vdash \mathsf{odd}(\mathsf{z})$. Applying the forward chaining rule reduces this to showing $\mathsf{odd}(\mathsf{s}(\mathsf{s}(n'))), \mathsf{odd}(n') \vdash \mathsf{odd}(\mathsf{z})$. But this follows by induction hypothesis plus weakening.

This establishes a weak form of completeness of the forward chaining program in the sense that if it saturates, then $\mathsf{odd}(\mathsf{z})$ must be present in the saturated database. A second argument shows that the database must always saturate (see Exercise 21.1), and therefore the forward chaining implementation is complete in the stronger sense of terminating and yielding a contradiction whenever n is even.

For the soundness direction we need to generalize the induction hypothesis, because $\mathsf{odd}(n)$ *true* $\vdash \mathsf{odd}(\mathsf{z})$ *true* will not match the situation even after a single step on the left. The problem is that a database such as $\mathsf{odd}(n), \mathsf{odd}(\mathsf{s}(n))$ will saturate and derive $\mathsf{odd}(\mathsf{z})$, but only one of the two numbers is even. Fortunately, it is sufficient to know that there exists some even number in the context, because we seed it with a singleton.[1]

[1] I am grateful to Deepak Garg for making this observation during lecture.

Lemma 21.2 *If* $\Gamma \vdash \mathsf{odd}(\mathsf{z})$ *where* Γ *consists of assumptions of the form* $\mathsf{odd}(\_)$*, then there exists an* $\mathsf{odd}(m) \in \Gamma$ *such that* $\mathsf{even}(m)$*.*

Proof: By induction on the structure of the given derivation $\mathcal{E}$.

Case: $\mathcal{E} = \dfrac{}{\Gamma', \mathsf{odd}(\mathsf{z}) \vdash \mathsf{odd}(\mathsf{z})}$ where $\Gamma = (\Gamma', \mathsf{odd}(\mathsf{z}))$.

$\mathsf{even}(\mathsf{z})$	By rule
Choose $m = \mathsf{z}$	$\mathsf{odd}(\mathsf{z}) \in \Gamma$

Case: $\mathcal{E} = \dfrac{\mathcal{E}'}{\Gamma', \mathsf{odd}(\mathsf{s}(\mathsf{s}(n))), \mathsf{odd}(n) \vdash \mathsf{odd}(\mathsf{z})}{\Gamma', \mathsf{odd}(\mathsf{s}(\mathsf{s}(n))) \vdash \mathsf{odd}(\mathsf{z})}$ where $\Gamma = (\Gamma', \mathsf{odd}(\mathsf{s}(\mathsf{s}(n))))$.

$\mathsf{even}(m')$ for some $\mathsf{odd}(m') \in (\Gamma', \mathsf{odd}(\mathsf{s}(\mathsf{s}(n))), \mathsf{odd}(n))$
By ind. hyp. on $\mathcal{E}'$

$\mathsf{odd}(m') \in (\Gamma', \mathsf{odd}(\mathsf{s}(\mathsf{s}(n))))$	Subcase
Choose $m = m'$	Since $\mathsf{odd}(m') \in \Gamma$

$\mathsf{odd}(m') = \mathsf{odd}(n)$	Subcase
$\mathsf{even}(n)$	By equality reasoning
$\mathsf{even}(\mathsf{s}(\mathsf{s}(n)))$	By rule
Choose $m = \mathsf{s}(\mathsf{s}(n))$	Since $\mathsf{odd}(\mathsf{s}(\mathsf{s}(n))) \in \Gamma$

□

21.3 Synchronous Atoms

When studying goal-directed search as the foundation of logic programming, we found that the notion of *focusing* gave us the right model for the search behavior of the connectives. Search is goal-directed if all the connectives are *asynchronous* so they can be decomposed eagerly as goals until an atomic goal is reached. Then we focus on one assumption and break this down until it matches the conclusion. The asynchronous fragment of intuitionistic logic is defined as follows.

$$A \quad ::= \quad P \mid A_1 \wedge A_2 \mid \top \mid A_2 \supset A_1 \mid \forall x.\, A$$

We summarize the rules of the focusing system in Figure 1. So far, has been the basis of backward chaining.

$$\frac{A \in \Gamma \quad \Gamma; A \ll P}{\Gamma \vdash P}\ \text{focus}L \qquad \frac{}{\Gamma; P \ll P}\ \text{id}R \qquad \begin{array}{c}\text{no rule for } P' \neq P \\ \Gamma; P' \ll P\end{array}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset R \qquad \frac{\Gamma; B \ll P \quad \Gamma \vdash A}{\Gamma; A \supset B \ll P}\ \supset L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge R \qquad \frac{\Gamma; A \ll P}{\Gamma; A \wedge B \ll P}\ \wedge L_1 \qquad \frac{\Delta; B \ll P}{\Gamma; A \wedge B \ll P}\ \wedge L_2$$

$$\frac{}{\Gamma \vdash \top}\ \top R \qquad \text{no } \top L \text{ rule}$$

$$\frac{\Gamma \vdash A \quad x \notin \text{FV}(\Gamma)}{\Gamma \vdash \forall x.\, A}\ \forall R \qquad \frac{\Delta; A(t/x) \ll P}{\Gamma; \forall x.\, A \ll P}\ \forall L$$

Figure 1: Focused Intuitionistic Logic; Asynchronous Fragment

Forward chaining violates the goal-directed nature of search, so we need to depart from the purely asynchronous fragment. Our change is minimalistic: we introduce only *synchronous atomic propositions* Q. For the sake of economy we also interpret inference rules

$$\frac{C_1\ \mathit{true} \ldots C_n\ \mathit{true}}{Q\ \mathit{true}}$$

without conjunction, writing them with iterated implication

$$\forall \mathbf{x}.\, C_1 \supset (C_2 \supset \ldots (C_n \supset Q))$$

instead. Here, C_i are all atoms, be they asynchronous (P) or synchronous (Q).

Since a goal can now be synchronous, we have the opportunity to focus on the right, if the right-hand side is a synchronous proposition (so far only Q). When a synchronous atomic proposition is in right focus, we succeed if the same proposition is in Γ; otherwise we fail.

$$\frac{\Gamma \gg Q}{\Gamma \vdash Q}\ \text{focus}R \qquad \frac{Q \in \Gamma}{\Gamma \gg Q}\ \text{id}L \qquad \begin{array}{c}\text{no rule for } Q \notin \Gamma \\ \Gamma \gg Q\end{array}$$

We also have to re-evaluate the left rule for implication.

$$\frac{\Gamma; B \ll P \quad \Gamma \vdash A}{\Gamma; A \supset B \ll P} \supset L$$

Strictly speaking, the focus should continue on both subformulas. However, when all propositions are asynchronous, we immediately lose right focus, so we short-circuited the step from $\Gamma \gg A$ to $\Gamma \vdash A$. Now that we have synchronous proposition, the rule needs to change to

$$\frac{\Gamma; B \ll P \quad \Gamma \gg A}{\Gamma; A \supset B \ll P} \supset L$$

and we add a rule

$$\frac{\Gamma \vdash A \quad A \neq Q}{\Gamma \gg A} \text{blur}R$$

A similar phenomenon arises on the left: when focused on a proposition that is asynchronous on the right (and not asynchronous on the left), we lose focus but we cannot fail as in the case of P.

$$\frac{\Gamma, Q \vdash P}{\Gamma; Q \ll A} \text{blur}L$$

Furthermore, all rules need to be generalized to allow either synchronous or asynchronous atoms on the right, which we write as C.

These considerations lead to the following rules, where have omitted the rules for conjunction, truth, and universal quantification. They are only changed in that the conclusion in the left rules can now be an arbitrary C, that is, a P or Q.

$$\frac{A \in \Gamma, A \neq Q \quad \Gamma; A \ll C}{\Gamma \vdash C} \text{focus}L \qquad \frac{}{\Gamma; P \ll P} \text{id}R \qquad \begin{array}{c}\text{no rule for } P \neq C \\ \Gamma; P \ll C\end{array}$$

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \supset R \qquad \frac{\Gamma; A_1 \ll C \quad \Gamma \gg A_2}{\Gamma; A_2 \supset A_1 \ll C} \supset L$$

$$\frac{\Gamma \gg Q}{\Gamma \vdash Q} \text{focus}R \qquad \frac{Q \in \Gamma}{\Gamma \gg Q} \text{id}L \qquad \begin{array}{c}\text{no rule for } Q \notin \Gamma \\ \Gamma \gg Q\end{array}$$

$$\frac{\Gamma \vdash A \quad A \neq Q}{\Gamma \gg A} \text{blur}R \qquad \frac{\Gamma, Q \vdash C}{\Gamma; Q \ll C} \text{blur}L$$

This system is sound and complete with respect to ordinary intuitionistic logic, no matter which predicates are designated as synchronous and asynchronous. As before, this can be proved via a theorem showing the admissibility of various forms of cut for focused derivations. However, it is important that all occurrences of a predicate have the same status, otherwise the system may become incomplete.

21.4 Pure Forward Chaining

In pure forward chaining all atomic predicates are considered synchronous. What kind of operational semantics can we assign to this system? In a situation $\Gamma \vdash A$ we first break down A (which, after all, is asynchronous) until we arrive at Q. At this point we have to prove $\Gamma \vdash Q$. There are two potentially applicable rules, focus$R$ and focus$L$. Let us consider these possibilities.

The focus$R$ rule will always be immediately preceded by id$L$, which would complete the derivation. So it comes down to a check if Q is in Γ.

The focus$L$ rule focuses on a program clause or fact in Γ. Consider the case of a propositional Horn clause $Q_1 \supset \ldots \supset Q_n \supset Q'$. We apply $\supset L$ rule, with premisses $\Gamma; Q_2 \supset \ldots \supset Q_n \supset Q' \ll C$ and $\Gamma \gg Q_1$. The only rule applicable to the second premiss is id$L$, so Q_1 must be in the database Γ. We continue this process and note that $Q_2, \ldots, Q_n$ must all be in the database Γ already. In the last step the first premiss is $\Gamma; Q' \ll Q$ which transitions to $\Gamma, Q' \vdash Q$.

In summary, applying a left focus rule to a clause $Q_1 \supset \ldots \supset Q_n \supset Q'$ reduces the sequent $\Gamma \vdash Q$ to $\Gamma, Q' \vdash Q$ if $Q_i \in \Gamma$ for $1 \leq i \leq n$. This is exactly the forward chaining step from before.

Overall, we can either right focus to see if the goal Q has already been proved, or apply a forward chaining step. A reasonable strategy is to saturate (repeated applying left focus until we make no more progress) and then apply right focus to see if Q has been deduced. The failure of left focus can be enforced by replacing the blur$L$ rule by

$$\frac{\Gamma, Q \vdash C \quad Q \notin \Gamma}{\Gamma; Q \ll C}\ \text{blur}L'.$$

This remains complete due to the admissibility of contraction, that is, if $\Gamma, A, A \vdash C$ then $\Gamma, A \vdash C$. This analysis also suggests that the left rule for implication is more perspicuous if written with the premisses reversed in

case the goal on the right-hand side is synchronous.

$$\frac{\Gamma \gg A_1 \quad \Gamma; A_2 \ll Q}{\Gamma; A_1 \supset A_2 \ll Q}\ \supset L$$

In the case of a Horn clause this means we first check if $A_1 = Q_1$ is in Γ and then proceed to analyze the rest of the clause.

21.5 Matching and Ground Bottom-Up Logic Programming

The analysis of forward chaining becomes only slightly more complicated if we allow the Horn clauses $\forall \mathbf{x}.\, Q_1 \supset \ldots \supset Q_n \supset Q'$ to be quantified as long as we restrict the variables in the head Q' of the clause to be a subset of the variables in $Q_1, \ldots, Q_n$. Then right focusing must return a substitution θ and we have the following rules, specialized to the Horn fragment with only synchronous atoms.

$$\frac{A \in \Gamma, A \neq Q' \quad \Gamma; A \ll Q}{\Gamma \vdash Q}\ \text{focus}L$$

$$\frac{\Gamma \gg Q_1 \mid \theta \quad \Gamma; A_2\theta \ll Q}{\Gamma; Q_1 \supset A_2 \ll Q}\ \supset L \qquad \frac{\Gamma; A(X/x) \ll Q \quad X \notin \text{FV}(\Gamma, A, Q)}{\Gamma; \forall x.\, A \ll Q}\ \forall L$$

$$\frac{\Gamma \gg Q \mid (\cdot)}{\Gamma \vdash Q}\ \text{focus}R \qquad \frac{Q' \in \Gamma \quad Q' = Q\theta}{\Gamma \gg Q \mid \theta}\ \text{id}L \qquad \begin{array}{c}\text{no rule if no such } Q', \theta \\ \Gamma \gg Q \mid \theta\end{array}$$

$$\frac{\Gamma, Q' \vdash Q \quad Q' \notin \Gamma}{\Gamma; Q' \ll Q}\ \text{blur}L'$$

By the restriction on free variables, the Q' in the blur$L$ cannot contain any free variables. The blur$R$ rule cannot apply in the Horn fragment. We also assumed for simplicity that the overall goal Q in $\Gamma \vdash Q$ is closed.

So on the Horn fragment under the common restriction that all variables in the head of a clause must appear in its body, bottom-up logic programming corresponds exactly to treating all atomic propositions as synchronous. The operational semantics requires matching, instead of full unification, because the database Γ consists only of ground facts.

21.6 Combining Forward and Backward Chaining

The focusing rules for the language given ealier (that is, all asynchronous connectives plus synchronous atoms) are sound and complete with respect

to the truth judgment and therefore a potential basis for combining forward and backward chaining. Backward chaining applies to asynchronous atoms and forward chaining to synchronous atoms. For the propositional case this is straightforward, but in the case of quantifiers it becomes difficult. We consider quantifiers in the next lecture and the propositional case briefly here, by example.

We look at the behavior of

$$C_1, C_1 \supset C_2, C_2 \supset C_3 \vdash C_3$$

for various assignment of C_1, C_2, and C_3 as synchronous or asynchronous. Interestingly, no matter what we do, in the focusing system there is exactly one proof of this sequent.

If all predicates are asynchronous,

$$P_1, P_1 \supset P_2, P_2 \supset P_3 \vdash P_3,$$

we must focus on $P_2 \supset P_3$. One step of backward chaining generates the subgoal

$$P_1, P_1 \supset P_2, P_2 \supset P_3 \vdash P_2.$$

Now we must focus on $P_1 \supset P_2$ and obtain the subgoal

$$P_1, P_1 \supset P_2, P_2 \supset P_3 \vdash P_1$$

which succeeds by focusing on P_1 on the left. No other proof paths are possible.

If all predicates are synchronous,

$$Q_1, Q_1 \supset Q_2, Q_2 \supset Q_3 \vdash Q_3,$$

we must focus on $Q_1 \supset Q_2$ because only Q_1 is directly available in the context. Focusing on Q_1 is prohibited because it is synchronous on the right, and therefore asynchronous on the left. We obtain the subgoal

$$Q_1, Q_2, Q_1 \supset Q_2, Q_2 \supset Q_3 \vdash Q_3.$$

Now we must focus on $Q_2 \supset Q_3$. Focusing on $Q_1 \supset Q_2$ would fail since Q_2 is already in the context, and focusing on Q_3 on the right would fail since Q_3 is not yet in the context. This second forward chaining step now generates

$$Q_1, Q_2, Q_3, Q_1 \supset Q_2, Q_2 \supset Q_3 \vdash Q_3.$$

At this point we can only focus on the right, which completes the proof.

Now consider a mixed situation

$$Q_1, Q_1 \supset P_2, P_2 \supset Q_3 \vdash Q_3.$$

Focusing on $Q_1 \supset P_2$ will fail, because the right-hand side does not match P_2. The only successful option is to focus on $P_2 \supset Q_3$ which generates two subgoals

$$Q_1, Q_1 \supset P_2, P_2 \supset Q_3 \vdash P_2$$

and

$$Q_1, Q_3, Q_1 \supset P_2, P_2 \supset Q_3 \vdash Q_3.$$

For the first we focus on $Q_1 \supset P_2$ and finish, for the second we focus on the right and complete the proof.

You are asked to consider other mixed situations in Exercise 21.2.

21.7 Beyond the Horn Fragment

The focusing rules are more general than the Horn fragment, but there is no particular difficulty in adopting the operational semantics as presented here, as long as we exclude the difficulties that arise due to quantification. In a later lecture we will consider adding other synchronous connectives (falsehood, disjunction, and existential quantification).

Here, we make only one remark about conjunction and truth. In case of intuitionistic logic they can safely be considered to be synchronous *and* asynchronous. This means we can add the rules

$$\frac{\Gamma \gg A_1 \quad \Gamma \gg A_2}{\Gamma \gg A_1 \wedge A_2} \qquad \frac{}{\Gamma \gg \top}$$

This allows us to write Horn clauses as $\forall \mathbf{x}.\, Q_1 \wedge \ldots \wedge Q_n \supset Q'$ without any change in the operational behavior compared with $\forall \mathbf{x}.\, Q_1 \supset \ldots \supset Q_n \supset Q'$.

We could similarly add some left rules, but this would require an additional judgment form to break down connectives that are asynchronous on the left, which we postpone to a later lecture.

21.8 Unification via Bottom-Up Logic Programming

We close this lecture with another example of bottom-up logic programming, namely and implementation of unification.

The implementation of unification by bottom-up logic programming illustrates a common type of program where saturation can be employed to great advantage. This is similar to the even example where we reason by contradication for a decidable predicate. Here, the predicate is *non-unifiability* of two terms s and t, as well as non-unifiability of sequences of terms $\mathbf{s}$ and $\mathbf{t}$ as an auxiliary predicate. In order to show that two terms are non-unifiable we assume they are unifiable, saturate, and then test the resulting saturated database for inconsistent information. We write $s \doteq t$ and $\mathbf{s} \doteq \mathbf{t}$ for the two equality relations.

Since there are a number of ways a contradiction can arise, we also introduce an explicit proposition contra to indicate a contradiction. In a later lecture we will see that we can in fact use $\bot$ with a sound logical interpretation, but that would requires us to go beyond the current setting.

The way to think about the rules is via the laws governing equality. We start with symmetry, transitivity, and reflexivity. Reflexivity actually gives us no new information (we already know the two terms are equal), so there is no forward rule for it.

$$\frac{s \doteq t}{t \doteq s} \qquad \frac{s \doteq t \quad t \doteq r}{s \doteq r}$$

Not all instances of transitivity are actually required (see Exercise 21.3); restricting it can lead to an improvement in the running time of the algorithm. Next, the congruence rules for constructors and sequences.

$$\frac{f(\mathbf{s}) \doteq f(\mathbf{t})}{\mathbf{s} \doteq \mathbf{t}} \qquad \frac{(s, \mathbf{s}) \doteq (t, \mathbf{t})}{s \doteq t} \qquad \frac{(s, \mathbf{s}) \doteq (t, \mathbf{t})}{\mathbf{s} \doteq \mathbf{t}} \qquad \begin{array}{c}(\cdot) \doteq (\cdot) \\ \text{no rule}\end{array}$$

There is no rule for $(\cdot) \doteq (\cdot)$ since this fact does not yield any new information. However, we have rules that note contradictory information by concluding contra.

$$\frac{f(\mathbf{s}) \doteq g(\mathbf{t}) \quad f \neq g}{\text{contra}} \qquad \frac{(\cdot) \doteq (t, \mathbf{t})}{\text{contra}} \qquad \frac{(s, \mathbf{s}) \doteq (\cdot)}{\text{contra}}$$

Even in the presence of variables, the rules so far will saturate, closing a given equality under its consequences. We consider $f(x, g(b)) \doteq f(a, g(x))$ as an example and show the generated consequences, omitting any identi-

ties $t \doteq t$ and intermediate steps relating a sequences to their elements.

$f(x, g(b)) \doteq f(a, g(x))$	Assumption
$f(a, g(x)) \doteq f(x, g(b))$	Symmetry
$x \doteq a$	Congruence
$g(b) \doteq g(x)$	Congruence
$a \doteq x$	Symmetry
$g(x) \doteq g(b)$	Symmetry
$b \doteq x$	Congruence
$x \doteq b$	Symmetry
$b \doteq a$	Transitivity
$a \doteq b$	Symmetry
contra	Clash $b \neq a$

The only point missing from the overall strategy to is to generate a contradiction due to a failure of the occurs-check. For this we have two new forms of propositions, $x \notin t$ and $x \notin \mathbf{t}$ which we use to propagate occurrence information.

$$\frac{x \doteq f(\mathbf{t})}{x \notin \mathbf{t}}$$

$$\frac{x \notin (t, \mathbf{t})}{x \notin t} \qquad \frac{x \notin (t, \mathbf{t})}{x \notin \mathbf{t}} \qquad \begin{array}{c} x \notin (\cdot) \\ \text{no rule} \end{array}$$

$$\frac{x \notin f(\mathbf{t})}{x \notin \mathbf{t}} \qquad \frac{x \notin x}{\mathsf{contra}} \qquad \begin{array}{c} x \notin y, x \neq y \\ \text{no rule} \end{array}$$

$$\frac{x \notin t \quad t \doteq s}{x \notin s}$$

The last rule is necessary so that, for example, the set $x \doteq f(y), y \doteq f(x)$ can be recognized as contradictory.

Let us apply the McAllester meta-complexity result. In the completed database, any two subterms of the original unification problem may be set equal, so we have $O(n^2)$ possibilities. Transitivity has $O(n^3)$ prefix firings, so a cursory analysis yields $O(n^3)$ complexity. This is better than the exponential complexity of Robinson's algorithm, but still far worse then the

linear time lower bound. Both the algorithm and the analysis can be refined in a number of ways. For example, we can restrict the uses of symmetry and transitivity to obtain better bounds, and we can postpone the use of non-occurrence to a second pass over a database saturated by the other rules.

This form of presentation of unification has become standard practice. It does not explicitly compute a unifier, but for unifiable terms it computes a kind of graph where the nodes in the original term (when viewed as a dag) are the nodes, and nodes are related by explicit equalities. From this a unifier can be read off by looking up the equivalence classes of the variables in the original unification problem.

Another nice property of unification, shared by many other saturation-based algorithms, is that it is *incremental*. This means that equations can be added one by one, and the database saturated every time, starting from the previously saturated one. If the equations ever become contradictory, contra is derived.

Here is a sketch how this might be used in the implementation of a logic programming engine. We have a constraint store, initially empty. Whenever logic programming search would call unification to obtain a unifier, we instead assume the equation into the database and saturate it. If we obtain a contradiction, unification fails and we have to backtrack. If not, we continue with the resulting constraint store. A neat thing about this implementation is that we never explicitly need to compute and apply a most general unifier: any goal we consider is always with respect to a saturated (and therefore consistent) set of equations.

21.9 Historical Notes

Although the notion that atoms may be synchronous or asynchronous, at the programmer's discretion, is relatively old [1], I believe that the observation connecting forward chaining to synchronous atoms is relatively recent [2], and was made in the setting of general theorem proving. An alternative approach to combining forward and backward chaining in logic programming using a monad [5] will be the subject of a later lecture.

The view of unification as a forward reasoning process similar to the one described here is due to Huet [3], although he maintained equivalence classes of terms much more efficiently than our naive specification, using the well-known union-find algorithm to arrive at an almost linear algorithm. Huet's basic idea was later refined by Martelli and Montanari [6] and in a different way by Paterson and Wegman [7] to obtain linear time

algorithms for unification.

The idea to handle unification problems via a store of constraints, to be updated and queried during computation, goes back to constraint logic programming [4]. It was elevated to logical status by Saraswat [8], although the connection to focusing, forward and backward chaining was not recognized at the time.

21.10 Exercises

Exercise 21.1 *Prove that the database initialized with* $\mathsf{odd}(n)$ *for some* n *and closed under forward application of the rule*

$$\frac{\mathsf{odd}(\mathsf{s}(\mathsf{s}(N)))}{\mathsf{odd}(N)}$$

will always saturate in a finite number of steps.

Exercise 21.2 *Consider two other ways to assign atoms to be synchronous or asynchronous for the sequent*

$$C_1, C_1 \supset C_2, C_2 \supset C_3 \vdash C_3$$

from Section 21.6 and show that there exists a unique proof in each case.

Exercise 21.3 *Improve the bottom-up unification algorithm by analyzing more carefully which instances of symmetry and transitivity are really needed. You may ignore the occurs-check, which we assume could be done in a second pass after the other rules saturate. What kind of McAllester complexity does your analysis yield?*

21.11 References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, pages 97–111, Seattle, Washington, August 2006. Springer LNCS 4130.

[3] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[4] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[5] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

[6] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[7] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[8] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1991. ACM Doctoral Dissertation Award Series.

15-819K: Logic Programming

Lecture 22

Hyperresolution

Frank Pfenning

November 14, 2006

In this lecture we lift the forward chaining calculus from the ground to the free variable case. The form of lifting required is quite different from the backward chaining calculus. For Horn logic, the result turns out to be hyperresolution.

22.1 Variables in Forward Chaining

Variables in backward chaining in the style of Prolog are placeholders for unknown terms. They are determined during unification, which occurs when the head of a program clause is compared to the current goal.

The same strategy does not seem appropriate for forward chaining. As a first example, consider

$$\forall x.\, Q(x) \vdash Q(a).$$

This violates the restrictions imposed for the last lecture, because x occurs in the head of $\forall x.\, Q(x)$ but not its body (which is empty).

We cannot focus on the right-hand side since no $Q(\_)$ is in the context. But we can focus on the assumption, $\forall x.\, Q(x)$. We have to guess a substitution term for x (which we will leave as a variable for now) and then blur focus.

$$\dfrac{\dfrac{\dfrac{\forall x.\, Q(x), Q(X) \vdash Q(a)}{\forall x.\, Q(x); Q(X) \ll Q(a)}\ \text{blur}L}{\forall x.\, Q(x); \forall x.\, Q(x) \ll Q(a)}\ \forall L}{\forall x.\, Q(x) \vdash Q(a)}\ \text{focus}L$$

Nothing within this focusing sequence gives us a clue to what X should be. If we now focus on the right, we can see in the second phase that X should be a in order to complete the proof.

But is the set $\forall x.\, Q(x), Q(X)$ actually saturated, or can we focus on $\forall x.\, Q(x)$ again? Since X is a placeholder, the proper interpretation of the sequent $\forall x.\, Q(x), Q(X) \vdash Q(a)$ should be:

There exists an X such that $\forall x.\, Q(x), Q(X) \vdash Q(a)$.

Now it could be we need two instances of the universal quantifier, so we might need to focus again on the left before anything else. An example of this is

$$\forall x.\, Q(x), Q(a) \supset Q(b) \supset Q'(c) \vdash Q'(c)$$

So a priori there is no local consideration to rule out focusing again on the left to obtain the context $\forall x.\, Q(x), Q(X), Q(Y)$, which is not redundant with $\forall x.\, Q(x), Q(X)$.

By extension of this argument we see that we cannot bound the number of times we need a universally quantified assumption. This means we can never definitively saturate the context. The existential interpretation of variables seems somehow incompatible with forward chaining and saturation.

22.2 Parametric Assumptions

Looking again at the deduction steps

$$\dfrac{\dfrac{\dfrac{\forall x.\, Q(x), Q(X) \vdash Q(a)}{\forall x.\, Q(x); Q(X) \ll Q(a)}\ \text{blur}L}{\forall x.\, Q(x); \forall x.\, Q(x) \ll Q(a)}\ \forall L}{\forall x.\, Q(x) \vdash Q(a)}\ \text{focus}L$$

we see that we lost a lot of information when blurring focus. We have actually obtain $Q(X)$ from the context without any restriction on what X is. In other words, we have really derived "*For any X, $Q(X)$*". When we added it back to the context, it became existentially quantified over the sequent: "*For some X, $\ldots, Q(X) \vdash \ldots$*".

It would make no sense to exploit this genericity of X by restoring the universal quantifier: we already know $\forall x.\, Q(x)$. Moreover, we would be introducing a connective during bottom-up reasoning which would violate all principles of proof search we have followed so far.

So we need to capture the fact that $Q(X)$ holds for any X in the form of a judgment. We write $\Delta \vdash Q$ where Δ is a context of variables. In a typed setting, Δ records the types of all the variables, but we ignore this slight generalization here. Now we have two forms of assumptions A and $\Delta \vdash Q$. We call the latter *parametric hypotheses* or *assumptions parametric in* Δ. There is no need to allow general parametric assumptions $\Delta \vdash A$, although research on contextual modal type theory suggests that this would be possible. The variables in Δ in a parametric assumption $\Delta \vdash Q$ should be considered bound variables with scope Q.

Parametric hypotheses are introduced in the blurring step.

$$\frac{\Gamma, (\Delta \vdash Q) \vdash Q' \quad \Delta = \mathrm{FV}(Q)}{\Gamma; Q \ll Q'}\ \mathrm{blur}L$$

We assume here that the sequent does not contain any free variables other than those in Q. Because for the moment we are only interested in forward chaining, this is a reasonable assumption. We discuss the issue of saturation below.

Now we consider the other rules of the focusing system, one by one, to see how to accomodate parametric hypotheses. We are restricting attention to the Horn fragment with only synchronous atoms.

We now rule out (non-parametric) assumptions Q and just allow $(\Delta \vdash Q)$. Closed assumptions Q are silently interpreted as $(\cdot \vdash Q)$.

The focus$L$ rule is as before: we can focus on any non-parametric A. By the syntactic restriction, this cannot be a Q, so we elide the side condition.

$$\frac{A \in \Gamma \quad \Gamma; A \ll Q}{\Gamma \vdash Q}\ \mathrm{focus}L$$

The implies$L$ rule also remains the same.

$$\frac{\Gamma \gg Q_1 \quad \Gamma; A_2 \ll Q}{\Gamma; Q_1 \supset A_2 \ll Q}\ \supset L$$

For the $\forall L$ rule we have to guess the substitution term t for x. This term t may contain some free variables that are abstracted in the blur step.

$$\frac{\Gamma; A(t/x) \ll Q}{\Gamma; \forall x.\, A \ll Q}\ \forall L$$

In the implementation t will be determined by unification, and we then abstract over the remaining free variables.

Focusing on the right is as before; the change appears in the identity rule

$$\frac{\Gamma \gg Q}{\Gamma \vdash Q}\ \text{focus}R \qquad \frac{(\Delta \vdash Q') \in \Gamma \quad Q'\theta = Q \quad \mathrm{dom}(\theta) = \Delta}{\Gamma \gg Q}\ \text{id}L$$

Right focus on Q still fails if there is no appropriate $(\Delta \vdash Q')$ and θ.

22.3 Unification and Generalization

As a next step, we make the unification that happens during forward chaining fully explicit. This is the natural extension of matching during ground forward chaining discussed in the last lecture.

$$\frac{A \in \Gamma \quad \Gamma; A \ll Q}{\Gamma \vdash Q}\ \text{focus}L$$

$$\frac{\Gamma \gg Q_1 \mid \theta \quad \Gamma; A_2\theta \ll Q}{\Gamma; Q_1 \supset A_2 \ll Q}\ \supset L \qquad \frac{\Gamma; A(X/x) \ll Q \quad X \notin \mathrm{FV}(\Gamma, A, Q)}{\Gamma; \forall x.\, A \ll Q}\ \forall L$$

$$\frac{\begin{array}{l}\rho \text{ renaming on } \Delta \\ (\Delta \vdash Q') \in \Gamma \quad Q'\rho \doteq Q \mid \theta\end{array}}{\Gamma \gg Q \mid \theta}\ \text{id}L \qquad \begin{array}{l}\text{no rule if no such } \Delta \vdash Q', \theta \\ \Gamma \gg Q \mid \theta\end{array}$$

$$\frac{\Gamma \gg Q \mid (\cdot)}{\Gamma \vdash Q}\ \text{focus}R \qquad \frac{\Gamma, (\Delta \vdash Q') \vdash Q \quad \Delta = \mathrm{FV}(Q')}{\Gamma; Q' \ll Q}\ \text{blur}L$$

We do not permit free variables in Q for a global goal $\Gamma \vdash Q$. This may be reasonable at least on the Horn fragment if the renaming ρ always chooses fresh variables, since during forward chaining we never focus on the right except to complete the proof.

Reconsider an earlier example to see this system in action.

$$\forall x.\, Q(x), Q(a) \supset Q(b) \supset Q'(c) \vdash Q'(c)$$

We must focus on $\forall x.\, Q(x)$, which adds $y \vdash Q(y)$ to the context.

$$\forall x.\, Q(x), Q(a) \supset Q(b) \supset Q'(c), (y \vdash Q(y)) \vdash Q'(c)$$

Now we can focus on the second assumption, using substitutions a/y and b/y for the two premisses and adding $\cdot \vdash Q'(c)$ to the context. Now we can focus on the right to prove $Q'(c)$.

22.4 Saturation via Subsumption

In the ground forward chaining system of the last lecture we characterized saturation by enforcing that a blur step must add something new to the context Γ.

$$\frac{\Gamma, Q \vdash Q_0 \quad Q \notin \Gamma}{\Gamma; Q \ll Q_0}\ \text{blur}L'$$

We must update this to account for parametric hypotheses $\Delta \vdash Q$. One should think of this as standing for an infinite number of ground assumptions, $Q\theta$ where $\mathrm{dom}(\theta) = \Delta$ and $\mathrm{cod}(\theta) = \emptyset$.

We can say that $(\Delta \vdash Q)$ adds nothing new to the context if every instance $Q\theta$ is already an instance of a parametric assumption Q'. That is, for every θ there exists $(\Delta' \vdash Q') \in \Gamma$ and θ' such that $Q'\theta' = Q\theta$. A tractable criterion for this is *subsumption*. We way that $(\Delta' \vdash Q')$ *subsumes* $(\Delta \vdash Q)$ if there exists a substitution θ' with $\mathrm{dom}(\theta') = \Delta'$ and $\mathrm{cod}(\theta') \subseteq \Delta$ such that $Q'\theta' = Q$. Then every instance $Q\theta$ is also an instance of Q' since $Q\theta = (Q'\theta')\theta = Q'(\theta'\theta)$.

The new blur rule then is

$$\frac{\Gamma, (\Delta \vdash Q) \vdash Q_0 \quad \text{no } (\Delta' \vdash Q') \in \Gamma \text{ subsumes } (\Delta \vdash Q)}{\Gamma; Q \ll Q_0}\ \text{blur}L'$$

As an example, if we have a theory such as

$$\forall x.\, \mathsf{pos}(\mathsf{s}(x)), \forall y.\, \mathsf{pos}(y) \supset \mathsf{pos}(\mathsf{s}(y))$$

where, in fact, the second clause is redundant, we will saturate quickly. After one step we assume $(w \vdash \mathsf{pos}(\mathsf{s}(w)))$. Now focusing on the first assumption will fail by subsumption, and focusing on the second will also fail by subsumption. After unification during forward chaining we have to ask if $(u \vdash \mathsf{pos}(\mathsf{s}(\mathsf{s}(u))))$ is subsumed before adding it to the context. But it is by the previous assumption, instantiating $\mathsf{s}(u)/w$. Therefore the above theory saturates correctly after one step.

Under this definition of saturation it is possible to represent a number of decision procedures as saturating forward chaining search with subsumption. In general, however, we are straying more from logic programming into general theorem proving

22.5 Beyond Saturation

In many applications saturation may be possible, but suboptimal in the sense that we would like to short-circuit and succeed as soon as possible.

An example is the program for unification in the previous lecture. As soon as we have a contradiction to the assumption that two terms are unifiable, we would like to stop forward-chaining. We can achieve this by adding another synchronous connective to the logic: falsehood ($\bot$).

As a conclusion, if we are focused on $\bot$ we fail. So, just as in backward search, $\bot$ as a goal represents failure.

As an assumption it is asynchronous, so we can succeed when we encounter it.

$$\frac{}{\Gamma; \bot \ll C}\ \bot L$$

There is an implicit phase transition here, from focusing on $\bot$ to asynchronously decomposing $\bot$ (which immediately succeeds).

In the unification example, the uses of contra can be replaced by $\bot$, which sometimes permits early success when a contradiction has been derived.

This device is also used in theorem proving where we don't expect to saturate, but hope to derive $\bot$ from the negation of a conjecture. At this point we have left logic programming and are firmly in the real of general purpose theorem proving: we no longer try to implement algorithms, but try to search for a proof in a general way.

If we restrict ourselves to the Horn fragment (though allowing $\bot$), and every atom is synchronous then the strategy of forward chaining with free variables presented here is also known as *hyperresolution* in the theorem proving literature.

Once we have the possibility to succeed by creating a contradiction, it is no longer necessary to have a relevant right-hand side. For example, instead of proving $\Gamma \vdash Q$ we can prove $\Gamma, Q \supset \bot \vdash \bot$ entirely by forward chaining on the left, without ever considering the right-hand side. Most of the classical resolution literature and even early presentations of logic programming use this style of presentation. The proof is entirely by contradiction, and there is not even a "right-hand side" as such, just a database of facts and rules Γ.

22.6 Splitting

If we also allow disjunction in the heads of clauses, but continue to force all atoms to be synchronous, we can represent what is known in the theorem proving literature as *splitting*. Since disjunction is asynchronous on the left, we need a new judgment form $\Gamma; A \vdash C$ where A is broken down

asynchronously. We transition into it when A is left asynchrounous, that is, Q, $\bot$, or $A_1 \vee A_2$. We give here the ground version.

$$\frac{\Gamma; A \vdash C \quad A = Q, \bot, A_1 \vee A_2}{\Gamma; A \ll C}\ \text{blur}L$$

$$\frac{}{\Gamma; \bot \vdash C}\ \bot L \qquad \frac{\Gamma; A_1 \vdash C \quad \Gamma; A_2 \vdash C}{\Gamma; A_1 \vee A_2 \vdash C}\ \vee L$$

$$\frac{\Gamma, A \vdash C \quad A \neq \bot, A_1 \vee A_2}{\Gamma; A \vdash C}$$

Unfortunately, this extension interacts poorly with free variables and parametric hypotheses. If there is a variable shared between A_1 and A_2 in the $\vee L$ rule, then it must be consistently instantiated on both sides and may not be abstracted. In the theorem proving context the rule is therefore restricted to cases where A_1 and A_2 share no free variables, which leaves the calculus complete for classical logic. Here, in intuitionistic logic, such a restriction would be incomplete.

When we move a formula A into Γ during decomposition of left asynchronous operators, we need to be able to abstract over its free variables even when the formula A is not atomic, further complicating the system.

To handle such situations we might allow existentially interpreted free variables in the context, and abstract only over those that are not free elsewhere in the sequent. However, then both subsumption and saturation become questionable again. It seems more research is required to design a larger fragment of intuitionistic logic that is amenable to a forward chaining operational semantics with reasonable saturation and subsumption behavior.

22.7 Historical Notes

Endowing assumptions with local contexts is a characteristic of contextual modal type theory [3] and the proof theory of the Nabla quantifier (∇) [2]. The former matches the use here, but is somewhat more general. The latter interprets the locally quantified variables as names subject to α conversion but cannot be instantiated by arbitrary terms.

There are a number of papers about using saturating hyperresolution as a decision procedure. A tutorial exposition and further references can be found in a chapter in the *Handbook of Automated Reasoning* [1].

22.8 References

[1] Christian Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 25, pages 1791–1849. Elsevier Science and MIT Press, 2001.

[2] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, October 2005.

[3] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. Submitted, September 2005.

15-819K: Logic Programming

Lecture 23

Linear Monadic Logic Programming

Frank Pfenning

November 16, 2006

In this lecture we extend the observations about forward and backward chaining from Horn logic to a rich set of linear connectives. In order to control the interaction between forward and backward chaining we use a *monad* to separate asynchronous from synchronous connectives. The result is the logic programming language LolliMon, which we illustrate with some examples.

23.1 Separating Asynchronous and Synchronous Connectives

The observations we have made about focusing, forward chaining, and backward chaining apply to various logics, including linear logic, which was indeed the origin of the notion of focusing. But it is difficult to combine forward and backward chaining and obtain a satisfactory operational semantics.

The problem can be broadly described as follows. First, if all connectives are asynchronous we obtain a clear and non-deterministically complete backward chaining semantics. Second, on the Horn fragment with only synchronous atoms we obtain a satisfactory forward chaining semantics with saturation. However, adding more synchronous connectives, or mixing synchronous with asynchronous connectives introduces a lot of uncontrolled non-determinism into the operational reading of programs. We approach general theorem proving, leaving the predictable operational behavior of Prolog behind.

We do not rule out that it may be possible to obtain a satisfactory semantics, but in this lecture we pursue a different path. We can control the interaction between asynchronous and synchronous connectives and

therefore between backward and forward chaining by creating a firewall between them called a *monad*. From the logical point of view a monad is a so-called *lax modal operator*. We will explain its laws below. For now we just note that we write $\{-\}$ for the modal operator.

In a complex linear logic proposition we can identify the places where we change from asynchronous to synchronous connectives or vice versa. At the first transition we require the explicit monadic firewall; the second transition is simply an inclusion. There is one exception, namely in a linear implication we switch sides (if the implication appears on the right the antecedent appears on the left), so we have to switch from asynchronous to synchronous propositions to remain in the same phase of decomposition.

$$\begin{array}{lrcl} \text{Asynch} & A & ::= & P \mid A_1 \mathbin{\&} A_2 \mid \top \mid S_1 \multimap A_2 \mid A_1 \supset A_2 \mid \forall x.\, A \mid \{S\} \\ \text{Synch} & S & ::= & Q \mid S_1 \oplus S_2 \mid \mathbf{0} \mid S_1 \otimes S_2 \mid \mathbf{1} \mid !A \mid \exists x.\, S \mid A \end{array}$$

Recall that P stands for asynchronous atoms and Q for synchronous atoms. We observe that the exponential modality of linear logic, !, has a special status: $!A$ is synchronous but the subformula does not continue the synchronous phase, but is asynchronous. Similarly, $A_1 \supset A_2$ is asynchronous, but A_1 in the antecedent which we might expect to be synchronous is in fact asynchronous. We will briefly explain this phenomenon later in the lecture.

We have been inclusive here, omitting only $S_1 \otimes! A_2$ and $\mathbf{s} \doteq \mathbf{t}$ which were necessary for residuation. The first is easy to add or define. Explicit equality is omitted here because in this lecture we concentrate on propositional and modal aspects of computation (see Exercise 23.2).

23.2 The Lax Modality

In order for the lax modality to have the right effect in separating the forward and backward chaining phases, we need to give it a proper logical foundation. In this section we just present this logical meaning; later we justify its role in the operational semantics. Since the semantics here is fully general, and does not depend on the division into asynchronous and synchronous operators, we just use B and C to stand for arbitrary proposition in linear logic, augmented by the lax modality.

In addition to the usual judgment $B\ true$ we have a new judgment on propositions, $B\ lax$. This can given be many different readings. Let us think of it for now as "*B is true subject to some (abstract) constraint*". This means

that $B\ lax$ is a *weaker* judgment then $B\ true$.

$$\frac{\Delta \Vdash B\ true}{\Delta \Vdash B\ lax}\ \text{lax}$$

This is a rule concerned only with the judgment, not with the proposition which remains unchanged. Of course, this rule is not invertible, or truth and lax truth would coincide. As in the prior sections on linear logic we omit the unrestricted context from the rules since it is generally just propagated from conclusion to premisses.

The second property is a form of the cut principle: if we can derive $B\ lax$ we are allowed to assume $B\ true$, but only if we are deriving a lax conclusion. This is so because deriving a lax conclusion permits a constraint.

If $\Delta_B \Vdash B\ lax$ *and* $\Delta_C, B\ true \Vdash C\ lax$ *then* $\Delta_C, \Delta_B \Vdash C\ lax$.

This should always hold, which means it is admissible when formulated as a rule of inference. This is not the case if we changed the conclusion to $C\ true$ since then we could derive the converse of the first rule, and truth and lax truth collapse.

The right and left rules for the lax modality are now straightforward.

$$\frac{\Delta \Vdash B\ lax}{\Delta \Vdash \{B\}\ true}\ \{-\}R \qquad \frac{\Delta, B\ true \Vdash C\ lax}{\Delta, \{B\}\ true \Vdash C\ lax}\ \{-\}L$$

Again, the conclusion of the $\{-\}L$ rule is restricted to the form $C\ lax$; allowing $C\ true$ here would be incorrect.

However, the other left rules in the sequent calculus must now permit an arbitrary judgment J on the right-hand side, which could be either $C\ true$ or $C\ lax$ where previously in could only be $C\ true$. The prior proof of the admissibility of cut and the identity principle can be easily extended to this fragment.

In terms of focusing, the modal operator looks at first a bit odd. In a deduction (read bottom-up), we go from $\{B\}\ true$ to $B\ lax$ and from there to $B\ true$.

On the right, the first transition is asynchronous: we can always strip the modal operator. The second transition is synchronous: we may have to wait for a left rule to be applied (specifically: to go from $\{C\}\ true$ to $C\ true$ for some C) before we can prove B true.

On the left, the first transition is synchronous: we cannot remove the modal operator until the right-hand side has the form $C\ lax$. The second

one, however, is asynchronous, because $B\ true$ is a stronger assumption than $B\ lax$.

We can also see this from the proof of the identity principle.

$$\dfrac{\dfrac{\dfrac{B\ true \Vdash B\ true}{B\ true \Vdash B\ lax}\ \text{lax}}{\{B\}\ true \Vdash B\ lax}\ \{-\}L}{\{B\}\ true \Vdash \{B\}\ true}\ \{-\}R$$

There is a "silent" transition on the left-hand side from $B\ lax$ to $B\ true$. Because the lax judgment is asynchronous as an assumption, we never need to consider it on the left-hand side.

23.3 The Exponential Modality Revisited

Above we have seen that the lax modality internalizes the judgment of lax truth, which is weaker than truth. Conversely, the exponential modality $!B$ of linear logic internalizes necessary truth (written $B\ valid$), which is stronger than truth. Recall first the judgmental rule, given here with a more general right hand-side J which could be $C\ true$ or $C\ lax$.

$$\dfrac{\Gamma, B\ valid; \Delta, B\ true \Vdash J}{\Gamma, B\ valid; \Delta \Vdash J}\ \text{copy}$$

The cut-like principle encodes that B is valid if it is true, without using any any truth assumptions.

If $\Gamma; \cdot \Vdash B\ true$ *and* $\Gamma, B\ valid; \Delta \Vdash J$ *then* $\Gamma; \Delta \Vdash J$.

When we internalize validity as a modal connective we obtain the following right and left rules.

$$\dfrac{\Gamma; \cdot \Vdash B\ true}{\Gamma; \cdot \Vdash\ !B\ true}\ !R \qquad \dfrac{\Gamma, B\ valid; \Delta \Vdash J}{\Gamma; \Delta, !B\ true \Vdash J}\ !L$$

We see that the left rule expresses that if $!B\ true$ then $B\ valid$. On the right-hand side, however, we have a silent transition from $!B\ true$, through $B\ valid$ to $B\ true$.

From this we can easily derive the focusing behavior. $!B$ is synchronous on the right, but the judgment $B\ valid$ is asynchronous. We therefore never need to explicitly consider $B\ valid$ as a conclusion.

Conversely, as an assumption $!B$ is asynchronous and can be decomposed eagerly to $B\ valid$. However, the assumption $B\ valid$ is synchronous (no matter what B is), so we need keep it as as assumption that we can focus on later.

This can also be seen from the proof of the identity principle.

$$\dfrac{\dfrac{\dfrac{B\ valid; B\ true \Vdash B\ true}{B\ valid; \cdot \Vdash B\ true}\ \text{copy}}{B\ valid; \cdot \Vdash\ !B\ true}\ !R}{\cdot; !B\ true \Vdash\ !B\ true}\ !L$$

In summary, the judgment forms $B\ lax$ and $B\ valid$ interact with focusing in the sense that $B\ lax$ is synchronous as a conclusion (no matter what B is) and $B\ valid$ is asynchronous as a conclusion (again, no matter what B is). This means $B\ valid$ need never explicitly be considered as a conclusion (it immediately becomes $B\ true$). Conversely $B\ valid$ is synchronous as an assumption and $B\ lax$ is asynchronous as an assumption. This emans $B\ lax$ need never explicitly be considered as an assumption (it immediately morphs into $B\ true$).

The asynchronous behaviors are only correct because of the modal restrictions: $B\ valid$ would only appear on the right if the linear context is empty, and $B\ lax$ would only appear on the left if the conclusion is $C\ lax$ for some C.

23.4 The Lax Modality and Search

We now preview the operational behavior of LolliMon, which can be divided into multiple phases derived from focusing. We will mostly build on the intuition for forward and backward chaining from the preceding lectures, except that linear forward chaining is more complex than just saturation.

Asynchronous goal decomposition. Starting with an asynchronous goal $A\ true$, we decompose it eagerly until we we come to either $\{S\}\ true$ or $P\ true$.

Backward chaining. If the goal is now of the form $P\ true$, we can focus on an assumption and decompose it. However, the ultimate head that we focus on must match P; if we reach $\{S\}$ instead we must fail because the

left rule for $\{-\}$ is not applicable when the conclusion is $P\ true$. This turns out to be a crucial advantage of having the lax modality, because if the head were an unprotected S we could not fail, but would now have to asynchronously decompose S.

Subgoals created during backward chaining are then solved in turn, as usual for backward chaining.

Forward chaining. If the goal on the right is $\{S\}\ true$ we exploit the fact that the lax modality is asynchronous and reduce it to $S\ lax$. At this point we could either focus on the left or focus on the right.

The operational semantics now prescribes a forward chaining phase. In the presence of linearity, this is intentially not complete (see the discussion of concurrency below). Nevertheless, we focus on an assumption and break it down until the head is either $P\ true$ or $\{S'\}\ true$. In the first case we fail, because we are focusing on $P\ true$ and it does not match the conclusion (which is lax). In the case $\{S'\}\ true$ we reduce it to $S'\ true$ which is then asynchronously decomposed until we can focus on a new assumption. The left rule for the lax modality is applicable here since the right-hand side is of the form $S\ lax$.

We continue forward chaining until we reach both *saturation* and *quiescence*. Saturation means that any forward step will not add any new assumption to Γ or Δ. Quiescence means we cannot focus on any linear assumption.

Once we have reached saturation and quiescence, focusing on the left is no longer possible, so we focus on the right, $S\ lax$, which becomes $S\ true$ under focus and is decomposed until we reach an asynchronous goal A to restart a new phase.

We now consider each phase in turn, writing out the relevant judgments and formal definition.

23.5 Asynchronous Goal Decomposition

We write this judgment as $\Gamma; \Delta; \Psi \Vdash A\ true$. Here, Γ is a context of unrestricted assumptions $A\ valid$, Δ is a context of linear assumptions $A\ true$, and Ψ is a context of ordered assumptions $S\ true$. Both A in the conclusion and the propositions S in Ψ are the ones that are decomposed. The context Ψ is ordered so we can restrict rules to operate on a unique proposition, removing any non-determinism from the rules.

$$\frac{\Gamma;\Delta;\cdot \Vdash A_1 \quad \Gamma;\Delta;\cdot \Vdash A_2}{\Gamma;\Delta;\cdot \Vdash A_1 \,\&\, A_2}\;\&R \qquad \frac{}{\Gamma;\Delta;\cdot \Vdash \top}\;\top R$$

$$\frac{\Gamma;\Delta;S_1 \Vdash A_2}{\Gamma;\Delta;\cdot \Vdash S_1 \multimap A_2}\;\multimap R \qquad \frac{\Gamma, A_1;\Delta;\cdot \Vdash A_2}{\Gamma;\Delta;\cdot \Vdash A_1 \supset A_2}\;\supset R$$

$$\frac{\Gamma;\Delta;\cdot \Vdash A \quad x \notin \mathrm{FV}(\Gamma,\Delta)}{\Gamma;\Delta;\cdot \Vdash \forall x.\, A}\;\forall R$$

$$\frac{\Gamma;\Delta;S_1,\Psi \Vdash P \quad \Gamma;\Delta;S_2,\Psi \Vdash P}{\Gamma;\Delta;S_1 \oplus S_2,\Psi \Vdash P}\;\oplus L \qquad \frac{}{\Gamma;\Delta;\mathbf{0},\Psi \Vdash P}\;\mathbf{0}L$$

$$\frac{\Gamma;\Delta;S_1,S_2,\Psi \Vdash P}{\Gamma;\Delta;S_1 \otimes S_2,\Psi \Vdash P}\;\otimes L \qquad \frac{\Gamma;\Delta;\Psi \Vdash P}{\Gamma;\Delta;\mathbf{1},\Psi \Vdash P}\;\mathbf{1}L$$

$$\frac{\Gamma, A;\Delta;\Psi \Vdash P}{\Gamma;\Delta;!A,\Psi \Vdash P}\;!L \qquad \frac{\Gamma;\Delta;S,\Psi \Vdash P \quad x \notin \mathrm{FV}(\Gamma,\Delta,P)}{\Gamma;\Delta;\exists x.\, S,\Psi \Vdash P}\;\exists L$$

$$\frac{\Gamma;\Delta,Q;\Psi \Vdash P}{\Gamma;\Delta;Q,\Psi \Vdash P}\;(Q)L \qquad \frac{\Gamma;\Delta,A;\Psi \Vdash P}{\Gamma;\Delta;A,\Psi \Vdash P}\;(A)L$$

$$\frac{\Gamma;\Delta \Vdash P}{\Gamma;\Delta;\cdot \Vdash P}\;(P)R \qquad \frac{\Gamma;\Delta \Vdash S\ \mathit{lax}}{\Gamma;\Delta;\cdot \Vdash \{S\}}\;\{-\}R$$

The last two rules transition to new judgments which represent so-called *neutral sequents* discussed in the next section.

23.6 Neutral Sequents

After an asynchronous decomposition has finished, we arrive at a neutral sequent. In LolliMon, there are two forms of neutral sequent, depending on whether the conclusion is true or lax. While the asynchronous decomposition is deterministic and can never fail, we now must make a choice about on which proposition to focus.

First, the case where the conclusion is $P\ \mathit{true}$. The possibilities for focusing initiate a backward chaining step.

$$\frac{\Gamma;\Delta;A \ll P \quad A \in \Gamma}{\Gamma;\Delta \Vdash P}\;\text{copy} \qquad \frac{\Gamma;\Delta;A \ll P}{\Gamma;\Delta,A \Vdash P}\;\text{focus}L \qquad \frac{\text{no focus}R\text{ rule}}{\Gamma;\Delta \Vdash P}$$

There is no way to focus on the right on an asynchronous atom since we only focus on synchronous propositions. Note that all propositions in Γ will be of the form A and therefore synchronous on the left. Similar, Δ consists of propositions that are synchronous on the left except for Q, which is analogous to allowing P on the right, and which may not be focused on.

When the conclusion is $S\ lax$ we can focus either on the left or on the right. Focusing on the left initiates a forward chaining step, focusing on the right terminates a sequence of forward chaining steps.

$$\frac{\Gamma; \Delta; A \ll S\ lax \quad A \in \Gamma}{\Gamma; \Delta \Vdash S\ lax}\ \text{copy} \qquad \frac{\Gamma; \Delta; A \ll S\ lax}{\Gamma; \Delta, A \Vdash S\ lax}\ \text{focus}L$$

$$\frac{\Gamma; \Delta \gg S\ true}{\Gamma; \Delta \Vdash S\ lax}\ \text{focus}R$$

We can see that a right-hand side $S\ lax$ means we are forward chaining, which we transition out of when when we focus in $S\ true$.

In a lower-level operational semantic specification we would add the precondition to the focus$R$ rule that no copy or focus$L$ rule is possible, indicating saturation and quiescence.

23.7 Backward Chaining

Backward chaining occurs when the right-hand side is $P\ true$ and we are focused on a proposition on the left. We refer to the right focus judgment for implications.

$$\frac{\Gamma; \Delta; A_1 \ll P}{\Gamma; \Delta; A_1 \mathbin{\&} A_2 \ll P}\ \&L_1 \qquad \frac{\Gamma; \Delta; A_2 \ll P}{\Gamma; \Delta; A_1 \mathbin{\&} A_2 \ll P}\ \&L_2 \qquad \begin{array}{c}\text{no } \top L \text{ rule} \\ \Gamma; \Delta; \top \ll P\end{array}$$

$$\frac{\Gamma; \Delta_1; A_1 \ll P \quad \Gamma; \Delta_2 \gg S_2}{\Gamma; \Delta_1, \Delta_2; S_2 \multimap A_1 \ll P} \multimap L \qquad \frac{\Gamma; \Delta; A_1 \ll P \quad \Gamma; \cdot \gg A_2}{\Gamma; \Delta; A_2 \supset A_1 \ll P} \supset L$$

$$\frac{\Gamma; \Delta; A(t/x) \ll P}{\Gamma; \Delta; \forall x.\, A \ll P}\ \forall L \qquad \begin{array}{c}\text{no } \{-\}L \text{ rule} \\ \Gamma; \Delta; \{S\} \ll P\end{array}$$

$$\frac{}{\Gamma; \Delta; P \ll P}\ (P)L \qquad \begin{array}{c}\text{not } (P)L \text{ rule if } P' \neq P \\ \Gamma; \Delta; P' \ll P\end{array}$$

Critical is the absence of the $\{-\}L$ rule: all forward chaining rules are disallowed during backward chaining. The logical rules for the lax modality

anticipate this: the right-hand side would have to have to be a lax judgment, but here it is truth. Without the monadic protection, this could not be done in a logically justified and complete way.

The rules for right focus also belong under this heading since they are invoked to terminate forward chaining or as a subgoal in backward chaining. When the right-hand side becomes asynchronous we transition back to the asynchronous decomposition judgment.

$$\frac{\Gamma; \Delta \gg S_1}{\Gamma; \Delta \gg S_1 \oplus S_2}\ \oplus R_1 \qquad \frac{\Gamma; \Delta \gg S_2}{\Gamma; \Delta \gg S_1 \oplus S_2}\ \oplus R_2 \qquad \begin{array}{c}\text{no } \mathbf{0}R \text{ rule} \\ \Gamma; \Delta \gg \mathbf{0}\end{array}$$

$$\frac{\Gamma; \Delta_1 \gg S_1 \quad \Gamma; \Delta_2 \gg S_2}{\Gamma; \Delta_1, \Delta_2 \gg S_1 \otimes S_2}\ \otimes R \qquad \frac{}{\Gamma; \cdot \gg \mathbf{1}}\ \mathbf{1}R$$

$$\frac{\Gamma; \cdot; \cdot \Vdash A}{\Gamma; \cdot \gg\ !A}\ !R \qquad \frac{\Gamma; \Delta \gg S(t/x)}{\Gamma; \Delta \gg \exists x.\, S}\ \exists R \qquad \frac{\Gamma; \Delta; \cdot \Vdash A}{\Gamma; \Delta \gg A}\ (A)R$$

$$\frac{}{\Gamma; Q \gg Q}\ (Q)R \qquad \begin{array}{c}\text{no rule for } \Delta \neq Q \\ \Gamma; \Delta \gg Q\end{array}$$

Focusing has strong failure conditions which are important to obtain predictable behavior. These are contained in the "missing rules" for the situations

$$\begin{array}{ll}\Gamma; \Delta; \{S\} \ll P\ true & \\ \Gamma; \Delta; P' \ll P\ true & \text{for } P' \neq P \\ \Gamma; \Delta \gg Q & \text{for } \Delta \neq Q\end{array}$$

all of which fail. The first is justified via the modal laws of lax logic, the second and third by properties of focusing on synchronous and asynchronous atoms. It is also important that we cannot focus on P on the right or Q on the left, which is again due to properties of focusing.

23.8 Forward Chaining

Forward chaining takes place when we focus on the left while the right-hand side is a lax judgment $S\ lax$. We can forward chain only if the ultimate head of the clause is a lax modality, which provides for a clean separation of the two phases with a logical foundation, and could not be easily justified otherwise as far as I can see.

The rules are mostly carbon copies of the left rules applied for forward chaining, except in the order of the premisses in the implication rules and,

of course, the rules for P and $\{S\}$.

$$\frac{\Gamma; \Delta; A_1 \ll S\ lax}{\Gamma; \Delta; A_1 \mathbin{\&} A_2 \ll S\ lax}\ \&L_1 \qquad \frac{\Gamma; \Delta; A_2 \ll S\ lax}{\Gamma; \Delta; A_1 \mathbin{\&} A_2 \ll S\ lax}\ \&L_2 \qquad \frac{\text{no } \top L \text{ rule}}{\Gamma; \Delta; \top \ll S\ lax}$$

$$\frac{\Gamma; \Delta_1 \gg S_1 \quad \Gamma; \Delta_1; A_2 \ll S\ lax}{\Gamma; \Delta_1, \Delta_2; S_1 \multimap A_2 \ll S\ lax}\ \multimap L \qquad \frac{\Gamma; \cdot \gg A_1 \quad \Gamma; \Delta; A_2 \ll S\ lax}{\Gamma; \Delta; A_1 \supset A_2 \ll S\ lax}\ \supset L$$

$$\frac{\Gamma; \Delta; A(t/x) \ll S\ lax}{\Gamma; \Delta; \forall x.\, A \ll S\ lax}\ \forall L \qquad \frac{\text{no } (P)L \text{ rule}}{\Gamma; \Delta; P \ll S\ lax}$$

$$\frac{\Gamma; \Delta; S' \Vdash S\ lax}{\Gamma; \Delta; \{S'\} \ll S\ lax}\ \{-\}L$$

We see $\Gamma; \Delta; P \ll S\ lax$ as an additional failure mode, due to focusing.

The rules for asynchronous decomposition of S' on the left once the focusing phase has been completed, are carbon copies of the rules when the conclusion is $P\ true$.

$$\frac{\Gamma; \Delta; S_1, \Psi \Vdash S\ lax \quad \Gamma; \Delta; S_2, \Psi \Vdash S\ lax}{\Gamma; \Delta; S_1 \oplus S_2, \Psi \Vdash S\ lax}\ \oplus L \qquad \frac{}{\Gamma; \Delta; \mathbf{0}, \Psi \Vdash S\ lax}\ \mathbf{0}L$$

$$\frac{\Gamma; \Delta; S_1, S_2, \Psi \Vdash S\ lax}{\Gamma; \Delta; S_1 \otimes S_2, \Psi \Vdash S\ lax}\ \otimes L \qquad \frac{\Gamma; \Delta; \Psi \Vdash S\ lax}{\Gamma; \Delta; \mathbf{1}, \Psi \Vdash S\ lax}\ \mathbf{1}L$$

$$\frac{\Gamma, A; \Delta; \Psi \Vdash S\ lax}{\Gamma; \Delta; !A, \Psi \Vdash S\ lax}\ !L \qquad \frac{\Gamma; \Delta; S', \Psi \Vdash S\ lax \quad x \notin \mathrm{FV}(\Gamma, \Delta, S)}{\Gamma; \Delta; \exists x.\, S', \Psi \Vdash S\ lax}\ \exists L$$

$$\frac{\Gamma; \Delta, Q; \Psi \Vdash S\ lax}{\Gamma; \Delta; Q, \Psi \Vdash S\ lax}\ (Q)L \qquad \frac{\Gamma; \Delta, A; \Psi \Vdash S\ lax}{\Gamma; \Delta; A, \Psi \Vdash S\ lax}\ (A)L$$

23.9 Backtracking and Committed Choice

The system from the previous sections is sound and complete when compared to intuitionistic linear logic with an added lax modality. This can be proved by a cut elimination argument as for other focusing systems.

Now we consider some lower-level aspects of the operational semantics. For backchaining with judgments $\Gamma; \Delta; \Psi \Vdash A$, $\Gamma; \Delta; A \ll P$ and $\Gamma; \Delta \gg S$, we solve subgoals from left to right, try alternatives from first-to-last, and backtrack upon failure. Moreoever, choices of terms t are post-

poned and determined by unification. So the backchaining fragment of LolliMon is fully consistent with pure Prolog and Lolli, that is, logic programming in the asynchronous fragment of linear logic. It is weakly complete which means that failure implies that a proof does not exist, but not all true propositions can be proven due to potentially non-terminating depth-first search.

For forward chaining, as defined by the judgments $\Gamma; \Delta; A \ll S\ lax$ and $\Gamma; \Delta; S' \Vdash S\ lax$ we use committed choice selection of the assumption to focus on. Moreoever, we continue to forward chain until we reach saturation and quiescence before we focus on the lax goal on the right. At present, we would consider it an error if we encounter a free logic variable during forward chaining because the semantics of this has not been satisfactorily settled.

If the program does not use the linear context in an essential way (that is, we are simulating unrestricted forward chaining in the linear setting), then due to the monotonicity of the unrestricted assumptions we still have non-deterministic completeness.

If the program *does* use the linear context then the system is complete in only a very weak sense: if we can always happen to make the right choice we can find a proof if it exists, but even if computation fails a proof might still exist because we do not backtrack.

I believe that this is actually the desired behavior because linear forward chaining was designed to model concurrency. The operational semantics of forward chaining corresponds exactly to the operational semantics of concurrent languages such as CCS or the π-calculus: if a transition is possible it may be taken without any chance for backtracking over this choice. This means programs are correct only if all legal computations behave correctly, such as, for example, avoiding deadlock. A corresponding correctness criteria applies to LolliMon program that use linear forward chaining: we must write the program in such a way that if multiple choices can be made, each one will give a correct answer in the end. The example in the next section illustrates this point.

As a programming language the logical fragment we have identified here is quite rich, since it allows backward chaining, saturating forward chaining, and linear forward chaining to capture many different sorts of operational behavior. We have to guard against viewing it as a general purpose inference engine. For example, it is often easy to write down a concurrent system as a linear LolliMon specification, but LolliMon presently provides little help in analyzing such a specification beyond simulating individual runs. This is an interesting area of further research.

23.10 Checking Bipartiteness on Graphs

The literature contains some examples of LolliMon programs, as does the LolliMon distribution. Many of the examples for forward chaining previously given in these notes such as CKY parsing or unification, can easily be expressed in LolliMon. We give here one more example, checking whether a given graph is bipartite, which illustrates a program that requires multiple saturating phases of forward chaining, and combines linear and nonlinear forward chaining.

A graph is bipartite if there is a division of its nodes into just two sets such that no two nodes in a set are connected to each other. Here is a high-level description of algorithm to check if a graph is bipartite. We use two colors, a and b to represent the two sets. We start with a graph where all nodes are unlabeled.

1. If all nodes are labeled, stop. The graph is bipartite.
2. Otherwise, pick an unlabeled node and color it a.
3. Propagate until no further changes occur to the graph:
   (a) If node u has color a and u is connected to w, color w with b.
   (b) If node u has color b and u is connected to w, color w with a.
4. If there is a node with both colors a and b the graph is not bipartite. Stop.
5. Otherwise, go to Step 1.

In the representation we have the following predicates, together with their intended usage.

$\mathsf{edge}(u, w)$	unrestricted; true if there is an edge from u to w
$\mathsf{unlabeled}(u)$	linear; true of node u if it has not yet been labeled
$\mathsf{label}(u, c)$	unrestricted; true if node u has color c
notbip	linear; true if graph is not bipartite

We start with the database initialized with unrestricted $\mathsf{edge}(u, w)$ and linear $\mathsf{unlabeled}(u)$ facts.

$$\mathsf{notbip} \multimap \exists U.\, !\mathsf{label}(U, \mathsf{a}) \otimes !\mathsf{label}(U, \mathsf{b}) \otimes \top.$$
$$\mathsf{notbip} \multimap \mathsf{unlabeled}(U) \otimes (\mathsf{label}(U, \mathsf{a}) \supset \{\mathsf{notbip}\}).$$

The first rule checks if there is a node with both colors and succeeds if that is the case. The second consumes an unlabeled node U, assigns it color a,

saturates by forward chaining, and then recurses, starting the next iteration of the algorithm. notbip fails if there is no two-colored node and no unlabeled node, in which case the graph is indeed bipartite.

The first two forward chaining rules are straightforward.

$$!\mathsf{label}(U, \mathsf{a}) \otimes !\mathsf{edge}(U, W) \multimap \{!\mathsf{label}(U, \mathsf{b})\}.$$
$$!\mathsf{label}(U, \mathsf{b}) \otimes !\mathsf{edge}(U, W) \multimap \{!\mathsf{label}(U, \mathsf{a})\}.$$

The third rule contains the interaction between linear and unrestricted assumptions: if a previously unlabeled node has been labeled, remove the assumption that it is unlabeled.

$$!\mathsf{label}(U, C) \otimes \mathsf{unlabeled}(U) \multimap \{\mathbf{1}\}.$$

Finally, a rule to take the symmetric closure of the edge relation, needed if we do not want to assume the relation is symmetric to start with.

$$!\mathsf{edge}(U, W) \multimap \{!\mathsf{edge}(W, U)\}.$$

Syntactically, $\mathsf{edge}(u, w)$, $\mathsf{label}(u, c)$, and notbip must be asynchronous atoms. Atoms $\mathsf{unlabeled}(u)$ could be either synchronous or asynchronous, and both posibilities execute correctly. A synchronous interpretation is most natural. There is a version of this program where label is also linear, in which case it would also be most naturally interpreted synchronously (see Exercise 23.4).

23.11 Historical Notes

The presentation of the lax modality in the form presented here originates with [3]; see that paper for further references on modal logic and monads.

LolliMon[1] [2] is a relatively recent development. The language supported by the implementation and described in the paper is slightly different from what we discussed here. In particular, all atoms are asynchronous, and nested implications in forward chaining are processed in reverse order to what we show here. Moreover, the implementation does not fully support $\oplus$ and $\mathbf{0}$, but its term language is a simply-typed λ-calculus with prefix polymorphism solved by pattern unification. One aspect of this is discussed in the next lecture.

An example applying LolliMon in computer security is given by Polakow and Skalka [4].

[1] Distribution available at `http://www.cs.cmu.edu/~fp/lollimon/`

LolliMon originated in the work on the Concurrent Logical Framework (CLF) [6, 1]. CLF is a type theory for formalizing deductive systems supporting state (through linearity) and concurrency (through a lax modality). Its original formulation lacks disjunction ($\oplus$) and falsehood ($\mathbf{0}$), but permits dependent types and a rich term language. The most recent and most complete work on the foundation of CLF is by Watkins [5].

23.12 Exercises

Exercise 23.1 *Extend the LolliMon language from this lecture, adding equality* $\mathbf{s} \doteq \mathbf{t}$ *and* $S \otimes! A$*, both of which are synchronous as goals and asynchronous as assumptions.*

Exercise 23.2 *Some compound connectives in LolliMon are redundant and could be eliminated. For example,* $A_1 \supset A_2$ *is equivalently expressed by* $(!A_1) \multimap A_2$*. Consider which connectives are redundant in this sense in the language.*

Exercise 23.3 *We have hinted in a prior lecture on resource management that, though logically equivalent, certain connectives may still be important for operational reasons. For example,* $S \otimes! A$ *and* $S \otimes (!A)$ *have different resource management behavior. Reconsider the redundant operators you identified in response to the Exercise 23.2 and determine which ones also have identical resource management properties and which do not.*

Exercise 23.4 *Rewrite the program to check whether graphs are bipartite, making the* label *predicate linear.*

On this program, consider the assignment where both unlabeled *and* label *are synchronous and the one where both are asynchronous. Explain differences in operational behavior, if any. Discuss which is more natural or potentially more efficient.*

23.13 References

[1] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[2] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declar-*

ative Programming (PPDP'05), pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

[3] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[4] Jeff Polakow and Christian Skalka. Specifying distributed trust management in LolliMon. In S.Zdancewic and V.R.Sreedhar, editors, *Proceedings of the Workshop on Programming Languages and Security*, Ottawa, Canada, June 2006. ACM.

[5] Kevin Watkins. CLF: A logical framework for concurrent systems. Thesis Proposal, May 2003.

[6] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

15-819K: Logic Programming

Lecture 24

Metavariables

Frank Pfenning

November 28, 2006

In this lecture we return to the treatment of logic variables. In Prolog and some extensions we have considered, logic variables are global, and equations involving logic variables are solved by unification. However, when universal goals $\forall x.\, A$ are allowed in backward chaining, or existential assumptions $\exists x.\, A$ in forward chaining, new parameters may be introduced into the proof search process. Ordinary unification on logic variables is now unsound, even with the occurs-check. We generalize logic variables to *metavariables*, a terminology borrowed from proof assistants and logical frameworks, and describe unification in this extended setting.

24.1 Parameters

When proving a universally quantified proposition we demand that proof to be *parametric*. In the rule this is enforced by requiring that x be new.

$$\frac{\Gamma; \Delta \Vdash A \quad x \notin \mathrm{FV}(\Gamma, \Delta)}{\Gamma; \Delta \Vdash \forall x.\, A}\ \forall R$$

The condition on x can always be satisfied by renaming the bound variable. Operationally, this means that we introduce a new parameter into the derivation when solving a goal $\forall x.\, A$.

We already exploited the parametricity of the derivation for the admissibility of cut by substituting a term t for x in the subderivation. The admissibility of cut, rewritten here for lax linear logic, has the following form, with J standing for either $C\ true$ or $C\ lax$:

$$\textit{If } \Gamma; \Delta_D \overset{\mathcal{D}}{\Vdash} A \textit{ and } \Gamma; \Delta_E, A \overset{\mathcal{E}}{\Vdash} J \textit{ then } \Gamma; \Delta_E, \Delta_D \overset{\mathcal{F}}{\Vdash} J.$$

Putting aside some issues related to the validity and lax judgments, this is proved by a nested induction, first on the structure of the formula A, then the two derivations $\mathcal{D}$ and $\mathcal{E}$. One of the critical cases for quantification:

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma; \Delta_D \Vdash A_1 \quad x \notin \mathrm{FV}(\Gamma, \Delta_D)\end{array}}{\Gamma; \Delta_D \Vdash \forall x.\, A_1}\, \forall R$ where $A = \forall x.\, A_1$ and

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Gamma; \Delta_E, A_1(t/x) \Vdash J\end{array}}{\Gamma; \Delta_E, \forall x.\, A_1 \Vdash J}\, \forall L.$$

$\Gamma; \Delta_D \Vdash A_1(t/x)$	By $\mathcal{D}_1(t/x)$, noting $x \notin \mathrm{FV}(\Gamma, \Delta_D)$
$\Gamma; \Delta_E, \Delta_D \Vdash J$	By ind.hyp. on $A_1(t/x)$, $\mathcal{D}_1(t/x)$, $\mathcal{E}_1$

The induction hypothesis applies in (first-order) lax linear logic because $A_1(t/x)$ may be considered a subformula of $\forall x.\, A_1$ since it contains fewer quantifiers and connectives. The condition $x \notin \mathrm{FV}(\Gamma, \Delta_D)$ is critical to guarantee that $\mathcal{D}_1(t/x)$ is a valid proof of $\Gamma; \Delta_D \Vdash A_1(t/x)$.

Parameters behave quite differently from logic variables in that during unification they may not be instantiated. Indeed, carrying out such a substitution would violate parametricity. But parameters interact with logic variables, which means that simply treating parameters as constants during unifications is unsound.

24.2 Parameter Dependency

We consider two examples, $\forall x.\, \exists y.\, x \doteq y$, which should obviously be true (pick x for y), and $\exists y.\, \forall x.\, x \doteq y$, which should obviously be false (in general, there is not a single y equal to all x).

We consider the proof search behavior in these two examples. First, the successful proof.

$$\dfrac{\dfrac{\dfrac{}{\Vdash x \doteq x}\, \doteq R}{\Vdash \exists y.\, x \doteq y}\, \exists R}{\Vdash \forall x.\, \exists y.\, x \doteq y}\, \forall R$$

With logic variables and unification, this becomes the following, assuming

we do not actually care to return the substitution.

$$\dfrac{\dfrac{\dfrac{x \doteq Y \mid (x/Y)}{\Vdash x \doteq Y}\doteq R}{\Vdash \exists y.\, x \doteq y}\exists R}{\Vdash \forall x.\, \exists y.\, x \doteq y}\forall R$$

Applying the substitution (x/Y) in the derivation leads to the first proof, which is indeed valid.

Second, the unsuccessful proof. In the calculus without logic variables we fail either because in the $\exists R$ step the substitution is capture-avoiding (so we cannot use x/y), or in the $\forall R$ step where we cannot rename x to y.

$$\dfrac{\dfrac{\begin{array}{c}\text{fails}\\ \Vdash x \doteq y\end{array}}{\Vdash \forall x.\, x \doteq y}\forall R}{\Vdash \exists y.\, \forall x.\, x \doteq y}\exists R$$

In the presence of free variables we can apparently succeed:

$$\dfrac{\dfrac{\dfrac{x \doteq Y \mid (x/Y)}{\Vdash x \doteq Y}\doteq R}{\Vdash \forall x.\, x \doteq Y}\forall R}{\Vdash \exists y.\, \forall x.\, x \doteq y}\exists R$$

However, applying the substitution (x/Y) into the derivation does not work, because Y occurs in the scope of a quantifier on x.

In order to prevent the second, erroneous solution, we need to prevent (x/Y) as a valid result of unification.

24.3 Skolemization

At a high level, there are essentially three methods for avoiding the above-mentioned unsoundness. The first, traditionally used in classical logics in a pre-processing phase, is *Skolemization*. In classical logic we usually negate the theorem and then try to derive a contradiction, in which case Skolemization has a natural interpretation. Given an assumption $\forall y.\, \exists x.\, A(x, y)$, for every y there exists an x such that $A(x, y)$ is true. This means that there must be a function f such that $\forall y.\, A(f(y), y)$ because f can simply select the appropriate x for every given y.

I don't know how to explain Skolemization in the direct, positive form except as a syntactic trick. If we replace universal quantifiers by a Skolem function of the existentials in whose scope it lies, then $\exists y.\, \forall x.\, x \doteq y$ is transformed to $\exists y.\, f(y) \doteq y$. Now if we pick an existential variable Y for y, then $f(Y)$ and Y are not unifiable due to the occurs-check.

Unfortunately, Skolemization is suspect for several reasons. In Prolog, there is no occurs-check, so it will not work directly. In logics with higher-order term languages, Skolemization creates a new function symbol f for every universal quantifier, which could be used incorrectly in other places. Finally, in intuitionistic logics, Skolemization can no longer be done in a preprocessing phase, although it can still be employed.

24.4 Raising

Raising is the idea that existential variables should *never* be allowed to depend on parameters. When confronted with an unsolvable problem such as $\exists y.\, \forall x.\, x \doteq y$ this is perfect.

However, when a solution does exist, as in $\forall x.\, \exists y.\, x \doteq y$ we need to somehow permit y to depend on x. We accomplish this by rotating the quantifier outward and turning it into an explicit function variable, as in $\exists y.\, \forall x.\, x \doteq y(x)$. Now y can be instantiated with the identity function $\lambda z.\, z$ to solve the equation. Note that y is does not contain any parameters as specified.

Raising works better than Skolemization, but it does require a term language allowing function variables. While this seems to raise difficult issues regarding unification, we can make the functional feature so weak that unification remains decidable and most general unifiers continue to exist. This restriction to so-called *higher-order patterns* stipulates that function variables be applied only to a list of distinct bound variables. In the example above this is the case: y is applies only to x. We briefly discuss this further in the section on *higher-order abstract syntax* below.

24.5 Contextual Metavariables

A third possibility is to record with every logic variable (that is, metavariable) the parameters it may depend on. We write $\Sigma \vdash X$ if the substitution term for X may depend on all the parameters in Σ. As we introduce parameters into a deduction we collect them in Σ. As we create metavariables, we collect them into another different context Θ, together with their contexts. We write $\Theta; \Sigma; \Gamma; \Delta \Vdash A$. No variable may be declared more than once. The

right rules for existential and universal quantifiers are:

$$\frac{\Theta; \Sigma, x; \Gamma; \Delta \Vdash A}{\Theta; \Sigma; \Gamma; \Delta \Vdash \forall x.\, A}\ \forall R \qquad \frac{\Theta, (\Sigma \vdash X); \Sigma; \Gamma; \Delta \Vdash A(X/x)}{\Theta; \Sigma; \Gamma; \Delta \Vdash \exists x.\, A}\ \exists R$$

By the convention that variables can be declared only once, we now omit the condition and use renaming to achieve freshness of X and x. Unification now also depends on Θ and Σ so we write $\Theta; \Sigma \vdash s \doteq t \mid \theta$.

Let us revisit the two examples above. First, the successful proof.

$$\dfrac{\dfrac{\dfrac{(x \vdash Y); x \vdash x \doteq Y \mid (x/Y)}{(x \vdash Y); x \Vdash x \doteq Y}\ \doteq R}{\cdot; x; \cdot; \cdot \Vdash \exists y.\, x \doteq y}\ \exists R}{\cdot; \cdot; \cdot; \cdot \Vdash \forall x. \exists y.\, x \doteq y}\ \forall R$$

The substitution in the last step is valid because Y is allowed to depend on x due to its declaration $x \vdash Y$.

In the failing example we have

$$\dfrac{\dfrac{\dfrac{\text{fails}}{(\cdot \vdash Y); x \vdash x \doteq Y \mid \_}}{(\cdot \vdash Y); x; \cdot; \cdot \Vdash x \doteq Y}\ \doteq R}{(\cdot \vdash Y); \cdot; \cdot; \cdot \Vdash \forall x.\, x \doteq Y}\ \forall R}{\cdot; \cdot; \cdot; \cdot \Vdash \exists y. \forall x.\, x \doteq y}\ \exists R$$

Now unification in the last step fails because the parameter x on the left-hand side is not allowed to occur in the substitution term for Y.

We call metavariables $\Sigma \vdash X$ *contextual metavariables* because they carry the context in which they were created.

24.6 Unification with Contextual Metavariables

The unification algorithm changes somewhat to account for the presence of parameters. The first idea is relatively straightforward: if $(\Sigma_X \vdash X)$ and we unify $X \doteq t$, then we fail if there is a parameter in t not in Σ_X.

But unification is a bit more subtle. Consider, for example, $(x \vdash X)$ and $(x, y \vdash Y)$ and the problem $X \doteq f(Y)$. In this case the substitution term for X may only depend on x, but not on y. But Y is allowed to depend on y, so just substituting $f(Y)/X$ would be unsound if later Y were instantiated

with y. So we need to restrict Y to depend only on x. In general, for a problem $X \doteq t$, we need to restrict any metavariable in t by intersecting its context with Σ_X.

Unfortunately, this means that unification must return a new Θ' as well as a substitution θ such that every free variable in the codomain of θ is declared in Θ. We write

$$\Theta; \Sigma \vdash t \doteq s \mid (\Theta' \vdash \theta)$$

for this unification judgment, and similarly for term sequences.

$$\frac{\mathbf{t} \doteq \mathbf{s} \mid (\Theta' \vdash \theta)}{\Theta; \Sigma \vdash f(\mathbf{t}) \doteq f(\mathbf{s}) \mid (\Theta' \vdash \theta)} \qquad \frac{x \in \Sigma}{\Theta; \Sigma \vdash x \doteq x \mid (\Theta' \vdash \cdot)}$$

$$\frac{\Theta; \Sigma \vdash t \doteq s \mid (\Theta_1 \vdash \theta_1) \quad \Theta_1; \Sigma \vdash \mathbf{t}\theta_1 \doteq \mathbf{s}\theta_1 \mid (\Theta_2 \vdash \theta_2)}{\Theta; \Sigma \vdash (t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid (\Theta_2 \vdash \theta_1\theta_2)}$$

$$\frac{}{\Theta; \Sigma \vdash (\cdot) \doteq (\cdot) \mid (\Theta \vdash \cdot)}$$

Second, the cases for metavariables. We fold the occurs-check into the restriction operation.

$$\frac{}{\Theta; \Sigma \vdash X \doteq X \mid (\Theta; \cdot)}$$

$$\frac{\Theta \vdash t|_X > \Theta'}{\Theta; \Sigma \vdash X \doteq t \mid (\Theta' \vdash t/X)} \qquad \frac{t = f(\mathbf{t}) \quad \Theta \vdash t|_X > \Theta'}{\Theta; \Sigma \vdash t \doteq X \mid (\Theta' \vdash t/X)}$$

Finally, the restriction operator:

$$\frac{(\Sigma_X \vdash X) \in \Theta \quad x \in \Sigma_X}{\Theta \vdash x|_X > \Theta} \qquad \begin{array}{c}\text{no rule for } \Sigma_X \vdash X, x \notin \Sigma_X \\ \Theta \vdash x|_X > \theta\end{array}$$

$$\frac{\Theta \vdash \mathbf{t}|_X > \Theta'}{\Theta \vdash f(\mathbf{t})|_X > \Theta'} \qquad \frac{}{\Theta \vdash (\cdot)|_X > \Theta} \qquad \frac{\Theta \vdash t|_X > \Theta_1 \quad \Theta_1 \vdash \mathbf{t}|_X > \Theta_2}{\Theta \vdash (t, \mathbf{t})|_X > \Theta_2}$$

$$\frac{X \neq Y; (\Sigma_X \vdash X) \in \Theta}{\Theta, (\Sigma_Y \vdash Y) \vdash Y|_X > \Theta, (\Sigma_Y \cap \Sigma_X \vdash Y)} \qquad \begin{array}{c}\text{no rule for} \\ \Theta \vdash X|_X > \_\end{array}$$

As indicated before, restriction can fail in two ways: in $x|_X$ if $x \notin \Sigma_X$ and when trying $X|_X$. The first we call a parameter dependency failure, the second an occurs-check failure. Overall, we also call restriction an *extended occurs-check*.

24.7 Types

One of the reasons to be so pedantic in the judgments above is the now straightforward generalization to the typed setting. The parameter context Σ contains the type declarations for all the variables, and declaration $\Sigma_X \vdash X : \tau$ contains all the types for the parameters that may occur in the substitution term for X. We can take this quite far to a dependent and polymorphic type theory and metavariables will continue to make sense. We only mention this here; details can be found in the literature cited below.

24.8 Higher-Order Abstract Syntax

It has been my goal in this class to present logic programming as a general paradigm of computation. It is my view that logic programming arises from the study of the structure of proofs (since computation is proof search) and that model-theoretic considerations are secondary. The liberation from the usual concerns about Herbrand models and classical reasoning has opened up a rich variety of new possibilities, including, for example, linear logic programming for stateful and concurrent systems.

At the same time I have been careful to keep my own interests in applications of logic programming in the background, and have drawn examples from a variety of domains such as simple list manipulation, algorithms on graphs, solitaire puzzles, decision procedures, dataflow analysis, etc. In the remainder of this lecture and the next one I will go into some examples of the use of logic programming in a logical framework, where the application domain itself also consists of logics and programming languages.

One of the pervasive notions in this setting is *variable binding*. The names of bound variables should not matter, and we should be able to substitute for them in a way that avoids capture. For example, in a proposition $\exists y.\, \forall x.\, x \doteq y$ we cannot substitute x for y because the binder on x would incorrectly *capture* the substitution term for y. Substitution into a quantified proposition is then subject to some conditions:

$$(\forall x.\, A)(t/y) = \forall x.\, A(t/y) \quad \text{provided } x \neq y \text{ and } x \notin \mathrm{FV}(t).$$

These conditions can always be satisfied by (usually silently) renaming bound variables, here x.

If we want to represent objects with variable binders (such as such as quantified propositions) as terms in a metalanguage, the question arises on how to represent bound variables. By far the most elegant means of accomplishing this is to represent them by corresponding bound variables in the metalanguage. This means, for example, that substitution in the object language is modeled accurately by substitution in the metalanguage without any additional overhead. This is the basic idea behind *higher-order abstract syntax*. A simple grammar decomposes terms into either *abstractions* or *applications*.

$$\begin{array}{lcll} \text{Abstractions} & b & ::= & x.\, b \mid t \\ \text{Applications} & t & ::= & h(b_1, \ldots, b_n) \\ \text{Heads} & h & ::= & x \mid f \end{array}$$

An abstraction $x.\, b$ binds the variable x with scope b. An application is just the term structure from first-order logic we have considered so far, except that the head of the term may be a variable as well as a function symbol, and the arguments are again abstractions.

These kinds of terms with abstractions are often written as λ-terms, using the notation $\lambda x.\, M$. However, here we do not use abstraction to form functions in the sense of functional programming, but simply to indicate variable binding. We therefore prefer to think of $\lambda x.\, M$ as $\lambda(x.\, M)$ and $\forall x.\, A$ as $\forall(x.\, A)$, clearly representing variable binding in each case and thinking of λ and $\forall$ as simple constructors.

We we substitute an abstraction for a variable, we may have to hereditarily perform substitution in order to obtain a term satisfying the above grammar. For example, if we have a term $\mathsf{lam}(x.\, E(x))$ and we substitute $(y.y)/E$ then we could not return $(y.y)\,(x)$ (which is not legal, according to our syntax), but substitute x for y in the body of the abstraction to obtain $\mathsf{lam}(x.\, x)$.

$$\begin{array}{l} (\mathsf{lam}(x.\, E(x)))((y.y)/E) \\ = \mathsf{lam}(x.\, (E(x))((y.y)/E)) \\ = \mathsf{lam}(x.\, y(x/y)) \\ = \mathsf{lam}(x.\, x) \end{array}$$

Since one form of substitution may engender another substitution on embedded terms we call it *hereditary substitution*.

A more detailed analysis of higher-order abstract syntax, hereditary substitution, and the interaction of these notions with typing is beyond the

scope of this course. We will use it in the next lecture in order to specify the operational semantics of a programming language.

24.9 Historical Notes

Traditionally, first-order theorem provers have used Skolemization, either statically (in classical logic), or dynamically (in non-classical logics) [13]. A clear presentation and solution of many issues connected to quantifier dependencies and unification has been given by Miller [5].

The idea of higher-order abstract syntax goes back to Church's type theory [1] in which all variable binding was reduced to λ-abstraction. Martin-Löf's system of arities (mostly unpublished) was a system of simple types including variable binding. Its important role in logical frameworks was identified by several groups in 1986 and 1987, largely independently, and with different applications: theorem proving [8], logic programming [6], and logical frameworks [2].

In programming languages, the idea of mapping bound variables in an object language to bound variables in a metalanguage is due to Huet and Lang [3]. Its use in programming environments was advocated and further analyzed by Elliott and myself [11]. This paper also coined the term "*higher-order abstract syntax*" and is therefore sometimes falsely credited with the invention of concept.

There are many operational issues raised by variable dependencies and higher-order abstract syntax, most immediately unification which is important in all three types of applications (theorem proving, logic programming, and logical frameworks). The key step is Miller's discovery of *higher-order patterns* [4] for which most general unifiers still exist. I generalized this latter to various type theories [9, 10].

The most recent development in this area is *contextual modal type theory* [7] which gives metavariables first-class status within a type theory, rather than consider them a purely operational artifact. A presentation of unification in a slighly simpler version of this type theory can be found in Pientka's thesis [12].

24.10 Exercises

Exercise 24.1 *Show an example that leads to unsoundness if parameter dependency is not respected during unification using only hereditary Harrop formulas, that it, the asynchronous fragment of intuitionistic logic.*

24.11 References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[3] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[4] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[5] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[6] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[7] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. Submitted, September 2005.

[8] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[9] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[10] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[11] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[12] Brigitte Pientka. *Tabled Higher-Order Logic Programming*. PhD thesis, Department of Computer Science, Carnegie Mellon University, December 2003. Available as Technical Report CMU-CS-03-185.

[13] N. Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, pages 522–536, Saratoga Springs, New York, June 1992. Springer-Verlag LNCS 607.

15-819K: Logic Programming

Lecture 25

Substructural Operational Semantics

Frank Pfenning

November 30, 2006

In this lecture we combine ideas from the previous two lectures, linear monadic logic programming and higher-order abstract syntax, to present a specification technique for programming languages we call *substructural operational semantics*. The main aim of this style of presentation is semantic modularity: we can add new language features without having to rewrite prior definitions for smaller language fragments. We determine that this is mostly the case, although structural properties of the specification such as weakening or contraction might change.

25.1 A Big-Step Natural Semantics

As a warm-up exercise, and also to understand the lack of modularity in traditional specifications, we present the semantics for functional abstraction and application in a call-by-value language. This is called *natural semantics* because of an analogy to *natural deduction*. The representation of terms employs higher-order abstract syntax, as sketched in the last lecture.

$$\text{Expressions} \quad e \quad ::= \quad x \mid \mathsf{lam}(x.\, e) \mid \mathsf{app}(e_1, e_2)$$

In the expression $\mathsf{lam}(x.\, e)$ the variable x is bound with scope e.

The main judgment is $e \hookrightarrow v$, where e and v are expressions. This is a big-step semantics, so the judgment directly relates e to its final value v.

$$\frac{}{\mathsf{lam}(x.\, e) \hookrightarrow \mathsf{lam}(x.\, e)} \qquad \frac{e_1 \hookrightarrow \mathsf{lam}(x.\, e_1') \quad e_2 \hookrightarrow v_2 \quad e_1'(v_2/x) \hookrightarrow v}{\mathsf{app}(e_1, e_2) \hookrightarrow v}$$

We represent $e \hookrightarrow v$ as $\mathsf{neval}(e, v)$. In the translation into a logic programming notation using higher order abstract syntax we have to be careful about variable binding. It would be incorrect to write the first rule as

$$\mathsf{neval}(\mathsf{lam}(x.\, E), \mathsf{lam}(x.\, E))$$

because E is (implicitly) quantified on the outside, so we could not instantiate with a term that contains x. Instead we must make any dependency explicit with the technique of raising from last lecture.

$$\mathsf{neval}(\mathsf{lam}(x.\, E(x)), \mathsf{lam}(x.\, E(x))).$$

Here, $E(x)$ is a term whose head is a variable. For the second rule we see how substitution is represented as application in the meta-language. E_1' will be bound to an abstraction $x.\, e_1'$, and $E_1'(V_2)$ will carry out the substitution of $e_1'(V_2/x)$.

$$\begin{array}{l}\mathsf{neval}(\mathsf{app}(E_1, E_2), V) \leftarrow \\ \quad \mathsf{neval}(E_1, \mathsf{lam}(x.\, E_1'(x))), \\ \quad \mathsf{neval}(E_1, V_2), \\ \quad \mathsf{neval}(E_1'(V_2), V).\end{array}$$

25.2 Substructural Operational Semantics

In a judgment $e \hookrightarrow v$ the whole state of execution must be present in the components of the judgment. This means, for example, when we add mutable store, we have to rewrite the judgment as $\langle s, e\rangle \hookrightarrow \langle s', v\rangle$, where s is the store before evaluation and s' after. Now all rules (including those for functions which should not be concerned with the store) have to be updated to account for the store. Similar considerations hold for continuations, exceptions, and other enrichments of the language.

Substructural operational semantics has an explicit goal to achieve a more modular presentation, where earlier rules may not have to be revisited. We achieve this through a combination of various ideas. One is the that logical rules that permit contexts are parametric in those contexts. For example, a left rule for conjunction

$$\frac{\Delta, A_1 \Vdash C\ \mathit{true}}{\Delta, A_1 \mathbin{\&} A_2 \Vdash C\ \mathit{true}}$$

remains valid even if new connectives or new judgments are added. The second idea is to evaluate expressions with explicit destinations whose nature remains abstract. Destinations are implemented as parameters.

Substructural operational semantics employs linear and unrestricted assumptions, although I have also considered ordered and affine assumptions. The conclusion on the right-hand is generally only relevant when we tie the semantics into a larger framework, so we omit it in an abstract presentation of the rules.

There are three basic propositions:

$\mathsf{eval}(e, d)$	Evaluate e with destination d
$\mathsf{comp}(f, d)$	Compute frame f with destination d
$\mathsf{value}(d, v)$	Value of destination d is v

Evaluation takes place asynchronously, following the structure of e. Frames are suspended computations waiting for a value to arrive at some destination before they are reawakened. Values of destinations, once computed, are like messages send to suspended frames.

In this first, pure call-by-value language, evaluations, computations, and values are all linear. All the rules are left rules, although we do not specify the right-hand side. A complete evaluation has the form

$$\begin{array}{c} \Delta, \mathsf{value}(d, v) \Vdash \\ \vdots \\ \Delta, \mathsf{eval}(e, d) \Vdash \end{array}$$

where v is the value of e. We begin with expressions $\mathsf{lam}(x.\, E(x))$ which are values and returned immediately to the destination D.

$$\frac{\Delta, \mathsf{value}(D, \mathsf{lam}(x.\, E(x))) \Vdash}{\Delta, \mathsf{eval}(\mathsf{lam}(x.\, E(x)), D) \Vdash}$$

In applications we evaluate the function part, creating a frame that remembers to evaluate the argument, once the function has been computed. For this, we need to create a new destination d_1. The derivation of the premiss must be parametric in d_1. We indicate this by labeling the rule itself with $[d_1]$.

$$\frac{\Delta, \mathsf{comp}(\mathsf{app}_1(d_1, E_2), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{app}(E_1, E_2), D) \Vdash}\ [d_1]$$

When the expression E_1 has been evaluated, we have to switch to evaluating the argument E_2 with a new destination d_2., keeping in mind that eventually we have to perform the function call.

$$\frac{\Delta, \mathsf{comp}(\mathsf{app}_2(V_1, d_2), D), \mathsf{eval}(E_2, d_2) \Vdash}{\Delta, \mathsf{comp}(\mathsf{app}(D_1, E_2), D), \mathsf{value}(D_1, V_1) \Vdash}\ [d_2]$$

It may be possible to reuse the destination D_1, but we are not interested here in this kind of optimization. It might also interfere negatively with extensibility later on if destinations are reused in this manner.

Finally, the β-reduction when both function and argument are known.

$$\frac{\Delta, \mathsf{eval}(E_1'(V_2), D) \Vdash}{\Delta, \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(x.\, E_1'(x)), D_2), D), \mathsf{value}(D_2, V_2) \Vdash}$$

25.3 Substructural Operational Semantics in LolliMon

It is easy to take the four rules of our substructural specification and implement them in LolliMon. We need here linear forward chaining and existential quantification to introduce new destinations. LolliMon's term language permits abstraction, so we can use this to implement higher-order abstract syntax.

$$\mathsf{eval}(\mathsf{lam}(x.\, E(x)), D) \multimap \{\mathsf{value}(D, \mathsf{lam}(x.\, E(x)))\}.$$

$$\mathsf{eval}(\mathsf{app}(E_1, E_2), D) \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{app}_1(d_1, E_2), D)\}.$$

$$\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{app}_1(D_1, E_2), D) \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes \mathsf{comp}(\mathsf{app}_2(V_1, d_2), D)\}.$$

$$\mathsf{value}(D_2, V_2) \otimes \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(x.\, E_1'(x)), D_2), D) \multimap \{\mathsf{eval}(E_1'(V_2), D)\}.$$

The only change we have made to the earlier specification is to exchange the order of eval, comp, and value propositions for a more natural threading of destinations.

LolliMon combines forward and backward chaining, so we can also write the top-level judgment to obtain the final value.

$$\mathsf{evaluate}(E, V) \circ\!\!- (\forall d_0.\, \mathsf{eval}(E, d_0) \multimap \{\mathsf{value}(d_0, V)\}).$$

25.4 Adding Mutable Store

We would now like to add mutable store to the operational semantics. We have three new kinds of expressions to create, read, and assign to a cell of mutable storage.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{ref}(e) \mid \mathsf{deref}(e) \mid \mathsf{assign}(e_1, e_2) \mid \mathsf{cell}(c)$$

There is also a new kind of value $\mathsf{cell}(c)$ where c is a destination which serves as a name for a cell of storage. Note that $\mathsf{cell}(c)$ cannot appear in the

source. In order to enforce this syntactically we would distinguish a type of values from the type of expressions, something we avoid here for the sake of brevity.

First, the rules for creating a new cell. I suggest reading these rules from last to first, in the way they will be used in a computation. We write the destinations that model cells at the left-hand side of the context. This is only a visual aid and has no logical significance.

$$\frac{\mathsf{value}(c_1, V_1), \Delta, \mathsf{value}(D, \mathsf{cell}(c_1)) \Vdash}{\Delta, \mathsf{comp}(\mathsf{ref}_1(D_1), D), \mathsf{value}(D_1, V_1) \Vdash}\ [c_1]$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{ref}_1(d_1), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{ref}(E_1), D) \Vdash}\ [d_1]$$

We keep track of the value of in a storage cell with an assumption $\mathsf{value}(c, v)$ where c is a destination and v is a value. While destinations to be used as cells are modeled here as linear, they are in reality *affine*, that is, they may be used at most once. The store, which is represented by the set of assumptions $\mathsf{value}(c_i, v_i)$ will remain until the end of the computation.

Next, reading the value of a cell. Again, read the rules from the bottom up, and the last rule first.

$$\frac{\mathsf{value}(C_1, V_1), \Delta, \mathsf{value}(D, V_1) \Vdash}{\mathsf{value}(C_1, V_1), \Delta, \mathsf{comp}(\mathsf{deref}_1(D_1), D), \mathsf{value}(D_1, \mathsf{cell}(C_1)) \Vdash}$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{deref}_1(d_1), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{deref}(E_1), D) \Vdash}\ [d_1]$$

Next, assigning a value to a cell. The assignment $\mathsf{assign}(e_1, e_2)$ returns the value of e_2.

$$\frac{\mathsf{value}(C_1, V_2), \mathsf{value}(D, V_2) \Vdash}{\mathsf{value}(C_1, V_1), \Delta, \mathsf{comp}(\mathsf{assign}_2(\mathsf{cell}(C_1), D_2), D), \mathsf{value}(D_2, V_2) \Vdash}$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{assign}_2(V_1, d_2), D), \mathsf{eval}(E_2, d_2) \Vdash}{\Delta, \mathsf{comp}(\mathsf{assign}_1(D_1, E_2), D), \mathsf{value}(D_1, V_1) \Vdash}\ [d_2]$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{assign}_1(d_1, E_2), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{assign}(E_1, E_2), D) \Vdash}\ [d_1]$$

Because values are included in expressions, we need one more rule for cells (which are treated as values). Even if they do not occur in expressions initially, the arise from substitutions of values into expressions.

$$\frac{\Delta, \mathsf{value}(D, \mathsf{cell}(C)) \Vdash}{\Delta, \mathsf{eval}(\mathsf{cell}(C), D) \Vdash}$$

All of these rules are just added to the previous rules for functions. We have achieved semantic modularity, at least for functions and store.

Again, it is easy to turn these rules into a LolliMon program.

$$\begin{array}{l}
\mathsf{eval}(\mathsf{ref}(E_1), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{ref}_1(d_1), D)\}. \\
\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{ref}_1(D_1), D) \\
\quad \multimap \{\exists c_1.\, \mathsf{value}(c_1, V_1) \otimes \mathsf{value}(D, \mathsf{cell}(c_1))\}. \\
\\
\mathsf{eval}(\mathsf{deref}(E_1), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{deref}_1(d_1), D)\}. \\
\mathsf{value}(D_1, \mathsf{cell}(C_1)) \otimes \mathsf{value}(C_1, V_1) \otimes \mathsf{comp}(\mathsf{deref}_1(D_1), D) \\
\quad \multimap \{\mathsf{value}(C_1, V_1) \otimes \mathsf{value}(D, V_1)\}. \\
\\
\mathsf{eval}(\mathsf{assign}(E_1, E_2), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{assign}_1(d_1, E_2), D)\}. \\
\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{assign}_1(D_1, E_2), D) \\
\quad \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes \mathsf{comp}(\mathsf{assign}_2(V_1, d_2), D)\}. \\
\mathsf{value}(D_2, V_2) \otimes \mathsf{comp}(\mathsf{assign}_2(\mathsf{cell}(C_1), D_2), D) \otimes \mathsf{value}(C_1, V_1) \\
\quad \multimap \{\mathsf{value}(C_1, V_2) \otimes \mathsf{value}(D, V_2)\}. \\
\\
\mathsf{eval}(\mathsf{cell}(C), D) \\
\quad \multimap \{\mathsf{value}(D, \mathsf{cell}(C))\}.
\end{array}$$

Written in SSOS form, the program evaluates e with some initial destination d_0 as

$$\begin{array}{c}
\Delta, \mathsf{value}(d_0, v) \Vdash \\
\vdots \\
\mathsf{eval}(e, d_0) \Vdash
\end{array}$$

where v is the value of e and

$$\Delta = \mathsf{value}(c_1, v_1), \ldots, \mathsf{value}(c_n, v_n)$$

for cells $c_1, \ldots, c_n$. The matter is complicated further by the fact that c_i, parameters introduced during the deduction, may appear in v. So we would have to traverse v to eliminate references to c_i, or we could just print it, or we could create some form of closure over the store Δ. In either case, we need to be sure to comsume Δ to retain linearity overall. If we just want to check termination, the top-level program would be

$$\mathsf{terminates}(E) \circ\!\!- \forall d_0.\, \mathsf{eval}(E, d_0) \multimap \{\exists V.\, \mathsf{value}(d_0, V) \otimes \top\}.$$

Here, the existential quantifier will be instantiated after forward chaining reaches quiescence, so it is allowed to depend on all the parameters introduced during forward chaining.

25.5 Adding Continuations

We now add callcc to capture the current continuation and throw to invoke a continuation as an example of an advanced control-flow construct.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{callcc}(x.\, e) \mid \mathsf{throw}(e_1, e_2) \mid \mathsf{cont}(d)$$

The intuitive meaning is that $\mathsf{callcc}(x.\, e)$ captures the current continuation (represented as the value $\mathsf{cont}(d)$) and substitutes it for x in e, and that $\mathsf{throw}(e_1, e_2)$ evaluates e_1 to v_1, e_2 to a continuation k and then invokes k on v_1.

In linear destination-passing style, we use a destination d to stand for a continuation. We invoke a continuation d on a value v simply by setting $\mathsf{value}(d, v)$. Any frame waiting to receive a value can be activated in this manner.

But this creates several problems in the semantics. To illustrate them, we add z and $\mathsf{s}(e)$ for zero and successor, and a new frame $\mathsf{s}_1(d)$ which waits to increment the value returned to destination d. See Exercise 25.2 for the rules.

Then an expression

$$\mathsf{s}(\mathsf{callcc}(k.\, \mathsf{s}(\mathsf{throw}(\mathsf{z}, k))))$$

evaluates to $\mathsf{s}(\mathsf{z})$, never returning anything to the inner frame waiting to calculate a successor. This means frames are no longer linear—they may be ignored and therefore be left over at the end.

But the problems do not end there. Consider, for example,

$$\mathsf{app}(\mathsf{callcc}(k.\, \mathsf{lam}(x.\, \mathsf{throw}(\mathsf{lam}(y.\, y), k))), \mathsf{z}).$$

Because any λ-expression is a value, the callcc returns immediately and applies the function $\mathsf{lam}(x.\ \ldots)$ to z. This causes the embedded throw to take place, this time applying $\mathsf{lam}(y.\ y)$ to z, yielding z as the final value. In this computation, the continuation in place when the first argument to app is evaluated is invoked twice: first, because we return to it, and then again when we throw to it. This means frames may not only be ignored, but also duplicated.

The solution is to make all frames $\mathsf{comp}(f, d)$ *unrestricted* throughout. At the same time the other predicates must remain linear: $\mathsf{eval}(e, d)$ so that there is only one thread of computation, and $\mathsf{value}(d, v)$ so that at any given time there is at most one value v at any given destination d.

We present the semantics for callcc and related constructs directly in LolliMon, which is more compact than the inference rule presentation.

$$\begin{array}{l}\mathsf{eval}(\mathsf{callcc}(k.\, E(k)), D) \\ \quad \multimap \{\mathsf{eval}(E(\mathsf{cont}(D)), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{eval}(\mathsf{throw}(E_1, E_2), D) \\ \quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes !\mathsf{comp}(\mathsf{throw}_1(d_1, E_2), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{value}(D_1, V_1) \otimes !\mathsf{comp}(\mathsf{throw}_1(D_1, E_2), D) \\ \quad \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes !\mathsf{comp}(\mathsf{throw}_2(V_1, d_2), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{value}(D_2, \mathsf{cont}(D_2')) \otimes !\mathsf{comp}(\mathsf{throw}_2(V_1, D_2), D) \\ \quad \multimap \{\mathsf{value}(D_2', V_1)\}.\end{array}$$

$$\begin{array}{l}\mathsf{eval}(\mathsf{cont}(D'), D) \\ \quad \multimap \{\mathsf{value}(D, \mathsf{cont}(D'))\}.\end{array}$$

Of course, all other rules so far must be modified to make the suspended computations (which we now recognize as continuations) unrestricted by prefixing each occurrence of $\mathsf{comp}(f, d)$ with '!'. The semantics is not quite modular in this sense.

With this change we have functions, mutable store, and continuations in the same semantics, specified in the form of a substructural operational semantics.

25.6 Substructural Properties Revisited

The only aspect of our specification we had to revise was the substructural property of suspended computations. If we step back we see that we can use substructural properties of various language features for a kind of taxonomy.

Our first observation is that for functions alone it would have been sufficient to keep the suspended computations next to each other and in order. In such an ordered specification we would not even have needed the destinations, because adjacency guarantees that values arrive at proper locations. We will leave this observation informal, rather than introducing a version of LolliMon with an ordered context, although this would clearly be possible.

If we add mutable store, then propositions $\mathsf{value}(d, v)$ remain linear, while propositions $\mathsf{value}(c, v)$ for destinations d that act as cells are affine. Continuations $\mathsf{comp}(f, d)$ are still linear, as are propositions $\mathsf{eval}(e, d)$.

If we further add a means to capture the continuation, then suspended computations $\mathsf{comp}(f, d)$ must become unrestricted because we may either ignore a continuation or return to it more than once. Values $\mathsf{value}(d, v)$ must remain linear, as must evaluations $\mathsf{eval}(e, d)$. Storage cells remain affine.

With sufficient foresight we could have made suspended computations $\mathsf{comp}(f, d)$ unrestricted to begin with. Nothing in the early semantics relies on their linearity. On other hand, it is more interesting to see what structural properties would and would not work for various languages, and also more natural to assume only the properties that are necessary.

25.7 Historical Notes

The presentation of a big-step operational semantics relating an expression to its value by inference rules is due to Kahn [2] under the name *natural semantics*. Earlier, Plotkin [6] developed a presentation of operational semantics using rewrite rules following the structure of expressions under the name *structural operational semantics* (SOS). I view substructural operational semantics as a further development and extension of SOS. Another generalization to achieve modularity is Mosses' modular structural operational semantics [4] which uses top-down logic programming and a form of row polymorphism for extensibility.

To my knowledge, the first presentation of an operational semantics in linear destination-passing style appeared as an example for the use of the Concurrent Logical Framework (CLF) [1]. The formulation there was intrinsically of higher order, which made reasoning about the rules more difficult. The approach was formulated as an independent technique for modular language specification in an invited talk [5], but only an abstract was published. Further examples of substructural operational semantics were given in the paper that introduced LolliMon [3].

Using ordered assumptions in logical frameworks and logic programming was proposed by Polakow and myself [8, 7], although this work did not anticipate monadic forward chaining as a computational mechanism.

25.8 Exercises

Exercise 25.1 *Prove that the natural semantics and substructural semantics for the functional fragment coincide in a suitable sense.*

Exercise 25.2 *Add natural number constants* z *and* $\mathsf{s}(e)$ *to our language specification to permit more examples for* callcc*. Give formulations both in substructural operational semantics and in LolliMon.*

Exercise 25.3 *Extend the functional language with unit, pairs, sums, void, and recursive types as well as recursion at the level of expressions. Give a substructural operational semantics directly in LolliMon.*

Exercise 25.4 *Think of other interesting control constructs, for example, for parallel or concurrent computation, and represent them in substructural operational semantics.*

Exercise 25.5 *Give a specification of a call-by-value language where we do not substitute a complete value for a term, but only the name for the destination which holds the (immutable) value. Which is the proper substructural property for such destinations?*

Further extend this idea to capture a call-by-need semantics where arguments are evaluated the first time they are needed and then memoized. This is the semantics underlying lazy functional languages such as Haskell.

25.9 References

[1] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[2] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

[3] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

[4] Peter D. Mosses. Foundations of modular SOS. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, pages 70–80, Szklarska Poreba, Poland, September 1999. Springer-Verlag LNCS 1672. Extended version available as BRICS Research Series RS-99-54, University of Aarhus.

[5] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In Wei-Ngan Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302.

[6] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[7] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

[8] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1999. Electronic Notes in Theoretical Computer Science, Volume 20.

15-819K: Logic Programming

Lecture 26

Datalog

Frank Pfenning

December 5, 2006

In this lecture we describe Datalog, a decidable fragment of Horn logic. Briefly, Datalog disallows function symbols, which means that the so-called Herbrand universe of ground instances of predicates is finite. Datalog has applications in databases and, more recently, in program analysis and related problems. We also sketch a promising new way to implement Datalog via its bottom-up semantics using BDDs to represent predicates.

26.1 Stratified Negation

Datalog arises from Horn logic via two restrictions and an extension. The most important restriction is to disallow function symbols: terms must be variables or be drawn from a fixed set of constant symbols. The second restriction is that any variable in the head of a clause also appears in the body. Together these mean that all predicates are decidable via a simple bottom-up, forward chaining semantics, since there are only finitely many propositions that can arise. These propositions form the so-called *Herbrand universe*.

If all domains of quantification are finite, we can actually drop the restriction on variables in clause heads, since a head such as $p(x)$ just stands for finitely many instances $p(c_1), \ldots, p(c_n)$, where $c_1, \ldots, c_n$ is an enumeration of the elements of the domain of p.

In either case, the restriction guarantees decidability. This means it is possible to add a sound form of constructive negation, called *stratified negation*. For predicates p and q we say p *directly depends on* q if the body of a clause with head $p(\mathbf{t})$ contains $q(\mathbf{s})$. We write $p \geq q$ for the reflexive and transitive closure of the direct dependency relation. If q does *not* depend

on p then we can decide any atom $q(\mathbf{s})$ without reference to the predicate p. This allows us to write clauses such as

$$p(\mathbf{t}) \leftarrow \ldots, \neg q(\mathbf{s}), \ldots$$

without ambiguity: first we can determine the extension of q and then conclude $\neg q(\mathbf{s})$ for ground term $\mathbf{s}$ if $q(\mathbf{s})$ was not found to be true.

If the domains are infinite, or we want to avoid potentially explosive expansion of schematics facts, we must slightly refine our restrictions from before: any goal $\neg q(\mathbf{s})$ should be such that $\mathbf{s}$ is ground when we have to decide it, so it can be implemented by a lookup assuming that q has already been saturated.

A Datalog program which is stratified in this sense can be saturated by sorting the predicates into a strict partial dependency order and then proceeding bottom-up, saturating all predicates lower in the order before moving on to predicates in a higher stratum.

Programs that are not stratified, such as

$$p \leftarrow \neg p.$$

or

$$p \leftarrow \neg q.$$
$$q \leftarrow \neg p.$$

do not have such a clear semantics and are therefore disallowed. However, many technical variations of the most basic one given above have been considered in the literature.

26.2 Transitive Closure

A typical use of Datalog is the computation of the transitive closure of a relation. We can also think of this as computing reachability in a directed graph given the definition of the edge relation.

In the terminology of Datalog, the *extensional data base* (EDB) is given by explicit (ground) propositions $p(\mathbf{t})$. The *intensional data base* (IDB) is given by Datalog rules, including possible stratified uses of negation.

In the graph reachability example, the EDB consists of propositions $\mathsf{edge}(x, y)$ for nodes x and y defining the edge relation. The path relation, which is the transitive closure of the edge relation, is defined by two rules which constitute the IDB.

$$\mathsf{path}(x, y) \leftarrow \mathsf{edge}(x, y).$$
$$\mathsf{path}(x, y) \leftarrow \mathsf{path}(x, z), \mathsf{path}(z, y).$$

26.3 Liveness Analysis, Revisited

As another example of the use of Datalog, we revisit the earlier problem of program analysis in a small imperative language.

$$\begin{array}{lcl} l & : & x = op(y, z) \\ l & : & \mathsf{if}\ x\ \mathsf{goto}\ k \\ l & : & \mathsf{goto}\ k \\ l & : & \mathsf{halt} \end{array}$$

We say a variable is *live* at a given program point l if its value will be read before it is written when computation reaches l. Following McAllester, we wrote a bottom-up logic program for liveness analysis and determined its complexity using prefix firings as $O(v{\cdot}n)$ where v is the number of variables and n the number of instructions in the program.

This time we take a different approach, mapping the problem to Datalog. The idea is to extract from the program propositions in the initial EDB of the following form:

- $\mathsf{read}(x, l)$. Variable x is read at line l.
- $\mathsf{write}(x, l)$. Variables x is written at line l.
- $\mathsf{succ}(l, k)$. Line k is a (potential) successor to line l.

The succ predicate depends on the control flow of the program so, for example, conditional jump instructions have more than one successor. In addition we will define by rules (and hence in the IDB) the predicate:

- $\mathsf{live}(x, l)$. Variable x may be live at line l.

Like most program analyses, this is a conservative approximation: we may conclude that a variable x is live at l, but it will never actually be read. On the other hand, if $\mathsf{live}(x, l)$ is not true, then we know for sure that x can never be live at l. This sort of information may be used by compilers in register allocation and optimizations.

First, we describe the extraction of the EDB from the program. Every program instruction expands into a set of assertions about the program

lines and program variables.

$$
\begin{array}{lcl}
l : x = op(y,z) & \leftrightarrow & \left\{ \begin{array}{l} \mathsf{read}(y,l) \\ \mathsf{read}(z,l) \\ \mathsf{write}(x,l) \\ \mathsf{succ}(l,l+1) \end{array} \right. \\
l : \mathsf{if}\ x\ \mathsf{goto}\ k & \leftrightarrow & \left\{ \begin{array}{l} \mathsf{read}(x,l) \\ \mathsf{succ}(l,k) \\ \mathsf{succ}(l,l+1) \end{array} \right. \\
l : \mathsf{goto}\ k & \leftrightarrow & \mathsf{succ}(l,k) \\
l : \mathsf{halt} & \leftrightarrow & \textit{none}
\end{array}
$$

Here we assume that the next line $l+1$ is computed explicitly at translation time.

Now the whole program analysis can be defined by just two Datalog rules.

$$
\begin{array}{lcl}
\mathsf{live}(w,l) & \leftarrow & \mathsf{read}(w,l). \\
\mathsf{live}(w,l) & \leftarrow & \mathsf{live}(w,k), \mathsf{succ}(k,l), \neg\mathsf{write}(w,l).
\end{array}
$$

The program is stratified in that live depends on read, write, and succ but not vice versa. Therefore the appeal to negation in the second clause is legitimate.

This is an extremely succinct and elegant expression of liveness analysis. Interestingly, it also provides a practical implementation as we will discuss in the remainder of this lecture.

26.4 Binary Decision Diagrams

There are essentially two "traditional" ways of implementing Datalog: one is by a bottom-up logic programming engine, the other using a top-down logic programming engine augmented with tabling in order to avoid non-termination. Recently, a new mechanism has been proposed using *binary decision diagrams*, which has been shown to be particularly effective in large scale program analysis.

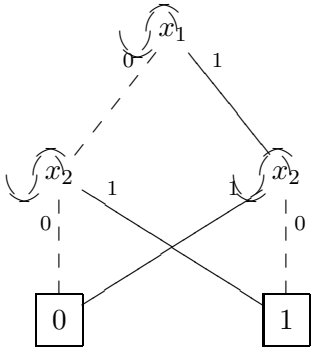
We briefly review here binary decision diagrams (BDDs). To be more precise, we will sketch the basics of *reduced ordered binary decision diagrams* (ROBDDs) which are most useful in this context for reasons we will illustrate below.

BDDs provide an often compact representation for Boolean functions. We will use this by viewing *predicates* as Boolean functions from the arguments (coded in binary) to either 1 (when the predicate is true) or 0 (when the predicate is false).

As an example consider the following Boolean function in two variables, x_1 and x_2.

$$\mathsf{xor}(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

We order the variables as x_1, x_2 and then present a diagram in which the variables are tested in the given order when read from top to bottom. When a variable is false (0) we follow the dashed line downward to the next variable, when it is true (1) we follow the solid line downward. When we reach the constant 0 or 1 we have determined the value of the Boolean function on the given argument values.

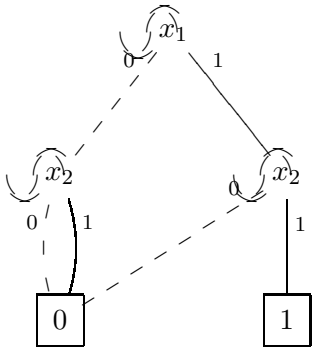


For example, to compute $\mathsf{xor}(1, 1)$ we start at x_1 follow the solid line to the right and then another solid line to the left, ending at 0 so $\mathsf{xor}(1, 1) = 0$.

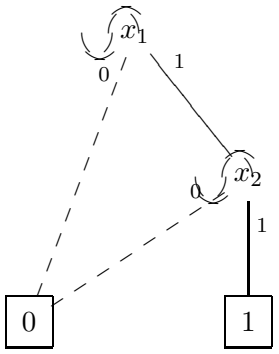
As a second simple example consider

$$\mathsf{and}(x_1, x_2) = x_1 \wedge x_2$$

If we make all choices explicit, in the given order of variables, we obtain

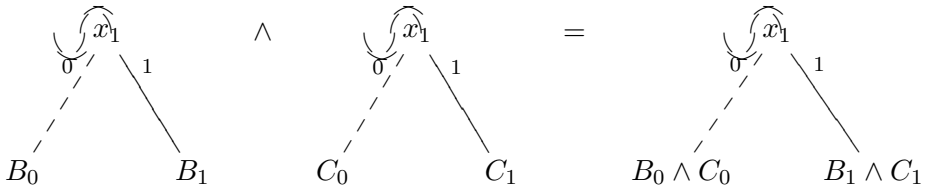


However, the test of x_2 is actually redundant, so we can simplify this to the following reduced diagram.



If we perform this reduction (avoiding unnecessary tests) and also share identical sub-BDDs rather than replicating them, then we call the OBDD *reduced*. Every Boolean function has a unique representation as an ROBDD once the variable order is fixed. This is one of the properties that will prove to be extremely important in the application of ROBDDs to Datalog.

Many operations on ROBDDs are straightforward and recursive, followed by reduction (at least conceptually). We will see some more examples later and now just consider conjunction. Assume we have two Boolean functions $B(x_1, \mathbf{x})$ and $C(x_1, \mathbf{x})$, where $\mathbf{x}$ represents the remaining variables. We notate $B(0, \mathbf{x}) = B_0$ and $B(1, \mathbf{x}) = B_1$ and similarly for C. We perform the following recursive computation



where the result may need to be reduced after the new BDD is formed. If the variable is absent from one of the sides we can mentally add a redundant node and then perform the operation as given above. On the leaves we have the equations

$$\boxed{0} \wedge B = B \wedge \boxed{0} = \boxed{0}$$

and

$$\boxed{1} \wedge B = B \wedge \boxed{1} = B$$

Other Boolean operations propagate in the same way; only the actions on the leaves are different in each case.

There are further operations which we sketch below in the example as we need them.

26.5 Datalog via ROBDDs

When implementing Datalog via ROBDDs we represent every predicate as a Boolean function. This is done in two steps: based on the type of the argument, we find out how many distinct constants can appear in this argument (say n) and then represent them with $\log_2(n)$ bits. The output of the function is always 1 or 0, depending on whether the predicate is true (1) or false (0). In this way we can represent the initial EDB as a collection of ROBDDs, one for each predicate.

Now we need to apply the rules in the IDB in a bottom-up manner until we have reached saturation. We achieve this by successive approximation, starting with the everywhere false predicate or some initial approximation based on the EDB. Then we compute the Boolean function corresponding to the body of each clause and combine it disjunctively with the current approximation. When the result turns out to be equal to the previous approximation we stop: saturation has been achieved. Fortunately this equality is easy to detect, since Boolean functions have unique representations.

We discuss the kind of operations required to compute the body of each clause only by example. Essentially, they are relabeling of variables, Boolean combinations such as conjunction and disjunction, and projection (which becomes an existential quantification).

As compared to traditional bottom-up strategy, where each fact is represented separately, we iterate over the whole current approximation of the predicate in each step. If the information is regular, this leads to a lot of sharing which can indeed be observed in practice.

26.6 An Example of Liveness Analysis

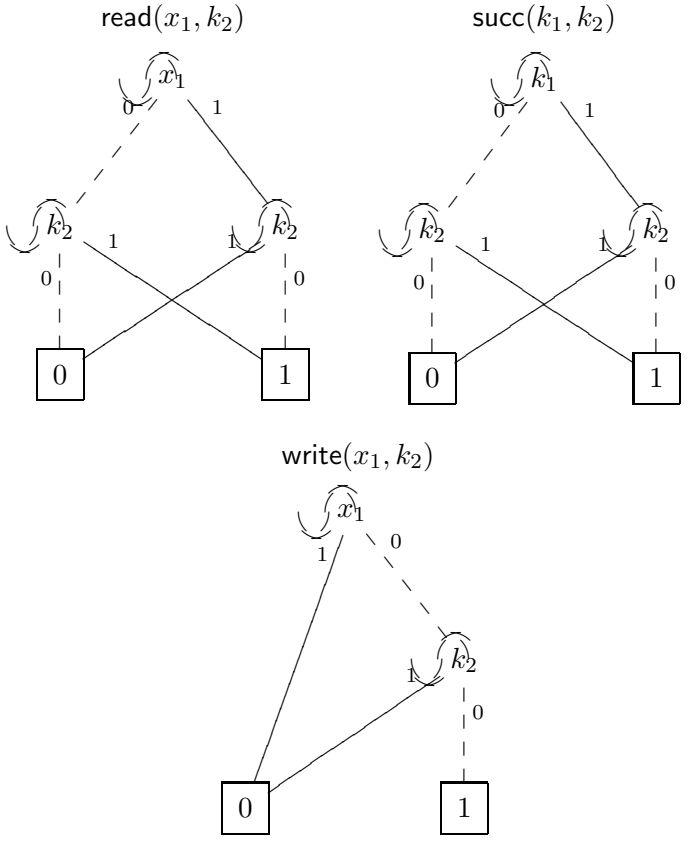
Now we walk through a small example of liveness analysis in detail in order to observe the BDD implementation of Datalog in action. Our program is very simple.

$$\begin{array}{lcl} l_0 & : & w_0 = w_1 + w_1 \\ l_1 & : & \mathsf{if}\ w_0\ \mathsf{goto}\ l_0 \\ l_2 & : & \mathsf{halt} \end{array}$$

To make things even simpler, we ignore line l_2 an analyse the liveness of the two variables w_0 and w_1 at the two lines l_0 and l_1. This allows us to represent both variables and program lines with a single bit each. We use 0 for l_0 and 1 for l_1 and similarly for the variables w_0 and w_1. Then the EDB we extract from the program is

$\mathsf{read}(1,0)$	we read w_1 at line l_0
$\mathsf{read}(0,1)$	we read w_0 at line l_1
$\mathsf{succ}(0,1)$	line l_1 succeeds l_0
$\mathsf{succ}(1,0)$	line l_0 succeeds l_1 (due to the goto)
$\mathsf{write}(0,0)$	we write to w_0 at line l_0

Represented as a BDD, these predicates of the EDB become the following three diagrams. We write x for arguments representing variables, k for arguments representing line numbers, and index them by their position in the order.

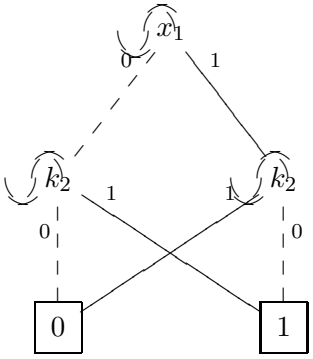


Now recall the IDB rules.

$$\begin{array}{lcl} \mathsf{live}(w,l) & \leftarrow & \mathsf{read}(w,l). \\ \mathsf{live}(w,l) & \leftarrow & \mathsf{live}(w,k), \mathsf{succ}(k,l), \neg\mathsf{write}(w,l). \end{array}$$

We initialize live with read, and use the notation $\mathsf{live}[0](x_1, k_2)$ to indicate it is the initial approximation of live.

$\mathsf{live}[0](x_1, k_2)$



This takes care of the first rule. The second rule is somewhat trickier, partly because of the recursion and partly because there is a variable on the right-hand side which does not occur on the left. This corresponds to an existential quantification, so to be explicit we write

$$\mathsf{live}(w,l) \leftarrow \exists k.\, \mathsf{live}(w,k), \mathsf{succ}(k,l), \neg\mathsf{write}(w,l).$$

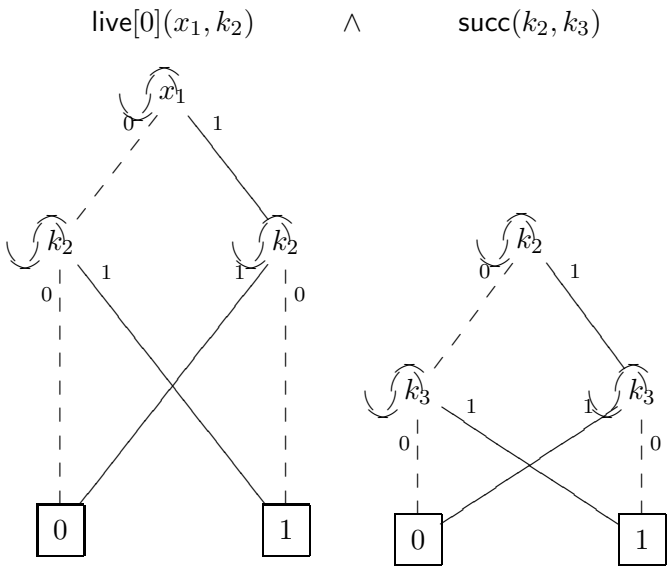
In order to compute the conjunction $\mathsf{live}[0](w,k) \wedge \mathsf{succ}(k,l)$ we need to relabel the variables so that the second argument for live is the same as the first argument for succ. To write the whole relabeled rule, also using logical notation for conjunction to emphasize the computation we have to perform:

$$\mathsf{live}(x_1,k_3) \leftarrow \exists k_2.\, \mathsf{live}(x_1,k_2) \wedge \mathsf{succ}(k_2,k_3) \wedge \neg\mathsf{write}(x_1,k_3).$$

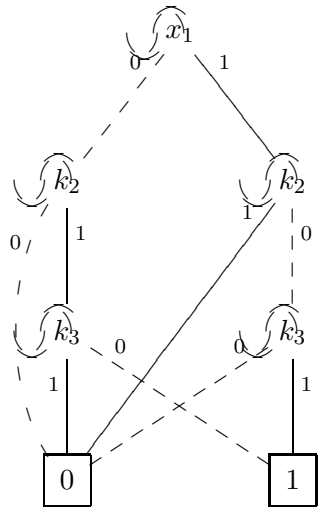
We now proceed to calculate the right hand side, given the fully computed definitions of succ and write and the initial approximation $\mathsf{live}[0]$. The intent is then to take the result and combine it disjunctively with $\mathsf{live}[0]$.

The first conjunction has the form $\mathsf{live}(x_1,k_2) \wedge \mathsf{succ}(k_2,k_3)$. Now, on each side one of the variables is not tested. We have lined up the corre-

sponding BDDs in order to represent this relabeling.



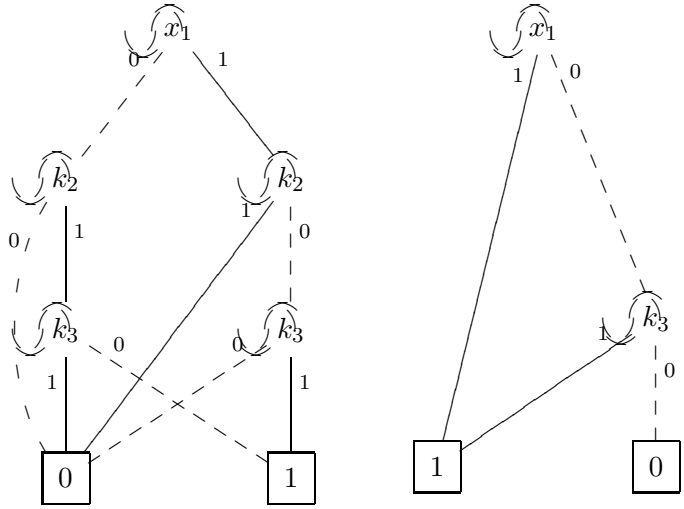
Computing the conjunction according to our recursive algorithm yields the following diagram. You are invited to carry out the computation by hand to verify that you understand this construction.



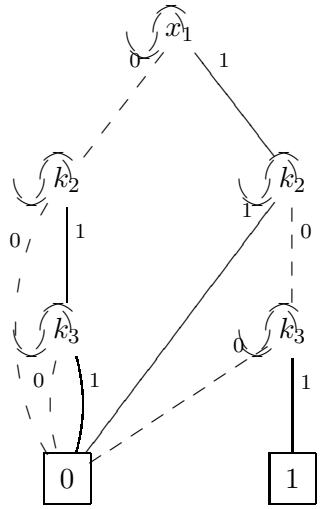
Now we have to conjoin the result with $\neg$write(x_1, k_3). We compute the negation simply by flipping the terminal 0 and 1 nodes at bottom of the

BDD, thus complementing the results of the Boolean function.

$$(\mathsf{live}[0](x_1, k_2) \wedge \mathsf{succ}(k_2, k_3)) \quad \wedge \quad \neg\mathsf{write}(x_1, k_3)$$



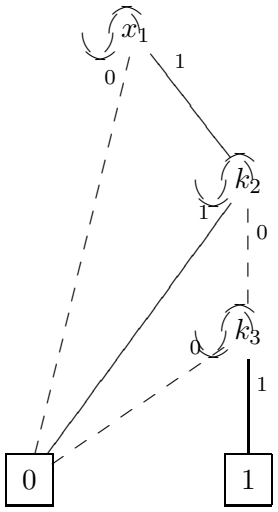
After the recursive computation we obtain the diagram below.



This diagram clearly contains some significant redundancies. First we notice that the test of k_3 on the left branch is redundant. Once we remove this node, the test of k_2 on the left-hand side also becomes redundant. In an efficient implementation this intermediate step would never have been computed in the first place, applying a technique such as hash-consing and immediately checking for cases such as the one here.

After removing both redundant tests, we obtain

$$\mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3)$$

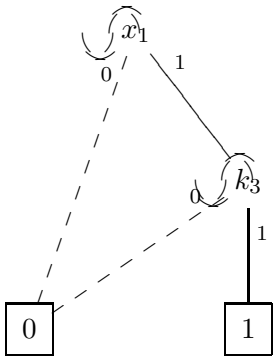


Recall that the right-hand side is

$$\exists k_2.\, \mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3).$$

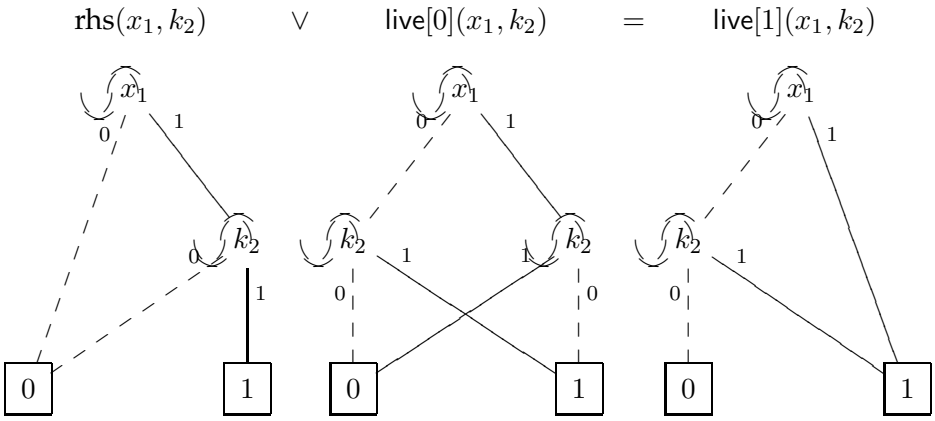
It remains to account for the quantification over k_2. In general, we compute $\exists x_1.\, B(x_1, \mathbf{x})$ as $B(0, \mathbf{x}) \vee B(1, \mathbf{x})$. In this example, the disjunction appears at the second level and is easy to compute.

$$\mathsf{rhs}(x_1, k_3) = \exists k_2.\, \mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3)$$



Now we rename again, k_3 to k_2 and disjoin it with $\mathsf{live}[0](x_1, k_2)$ to obtain

$\mathsf{live}[1](x_1, k_2)$.



At this point we have gone through one iteration of the definition of live. Doing it one more time actually does not change the definition any more ($\mathsf{live}[2](x_1, k_2) = \mathsf{live}[1](x_1, k_2)$) so the database has reached saturation (see Exercise 26.1).

Let us interpret the result in terms of the original program.

$$\begin{array}{lcl} l_0 & : & w_0 = w_1 + w_1 \\ l_1 & : & \mathsf{if}\ w_0\ \mathsf{goto}\ l_0 \\ l_2 & : & \mathsf{halt} \end{array}$$

The line from x_1 to 1 says that variable w_1 is live at both locations (we do not even test the location), which we can see is correct by examining the program. The path from $x_1 = 0$ through $k_2 = 1$ to 1 states that variable w_0 is live at l_1, which is also correct since we read its value to determine whether to jump to l_0. Finally, the path from $x_1 = 0$ through $k_2 = 0$ to 0 encodes that variables w_0 is not live at l_0, which is also true since l_0 does not read w_1, but writes to it. Turning this information into an explicit database form, we have derived

$$\begin{array}{l} \mathsf{live}(w_1, l_0) \\ \mathsf{live}(w_1, l_1) \\ \mathsf{live}(w_0, l_1) \\ \neg\mathsf{live}(w_0, l_0) \quad \textit{(implicitly)} \end{array}$$

where the last line would not be explicitly shown but follows from its absence in the saturated state.

While this may seem very tedious even in this small example, it has in fact shown itself to be quite efficient even for very large programs. However, a number of optimization are necessary to achieve this efficiency, as mentioned in the papers cited below.

26.7 Prospectus

BDDs were successfully employed in model checking for finite state systems. In our setting, this would correspond to a linear forward chaining process where only the rules are unrestricted and facts are linear and thus subject to change on each iteration. Moreover, the states (represented as linear contexts) would have to satisfy some additional invariants (for example, that each proposition occurs at most once in a context).

The research on Datalog has shown that we can use BDDs effectively for saturating forward chaining computations. We believe this can be generalized beyond Datalog if the forward chaining rules have a subterm property so that the whole search space remains finite. We only have to find a binary coding of all terms in the search space so that the BDD representation technique can be applied.

This strongly suggests that we could implement an interesting fragment of LolliMon which would encompass both Datalog and some linear logic programs subject to model checking, using BDDs as a uniform engine.

26.8 Historical Notes

The first use of deduction in databases is usually ascribed to a paper by Green and Raphael [3] in 1968, which already employed a form of resolution. The connection between logic programming, logic, and databases became firmly established during a workshop in Toulouse in 1977; selected papers were subsequently published in book form [2] which contained several seminal papers in logic programming. The name *Datalog* was not coined until the 1980's.

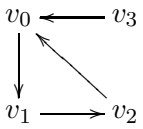
BDDs were first proposed by Randy Bryant [1]. Their use for implementing Datalog to statically analyze large programs was proposed and shown to be effective by John Whaley and collaborators [4], with a number of more specialized and further papers we do not cite here. The resulting system, `bddbddb` (BDD-Based Deductive DataBase) is available on SourceForge[1].

[1] `http://bddbddb.sourceforge.net/`

26.9 Exercises

Exercise 26.1 *Show all the intermediate steps in the iteration from* $\mathsf{live}[1](x_1, k_2)$ *to* $\mathsf{live}[2](x_1, k_2)$ *and confirm that saturation has been reached.*

Exercise 26.2 *Consider a 4-vertex directed graph*



Represent the edge *relation as a BDD and saturate the database using the two rules for transitive closure*

$$\mathsf{path}(x, y) \leftarrow \mathsf{edge}(x, y).$$
$$\mathsf{path}(x, y) \leftarrow \mathsf{path}(x, z), \mathsf{path}(z, y).$$

similer to the way we developed the liveness analysis example. Note that there are 4 vertices so they must be coded with 2 bits, which means that edge *and* path *each have 4 boolean arguments, two bits for each vertex argument.*

Exercise 26.3 *Give an encoding of another program analysis problem by showing (a) the extraction procedure to construct the initial EDB from the program, and (b) the rules defining the EDB.*

26.10 References

[1] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[2] Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases*. Plenum Press, 1978. Edited proceedings from a workshop in Toulouse in 1977.

[3] C. Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 23rd ACM National Conference*, pages 169–181, Washington, D.C., August 1968. ACM Press.

[4] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 97–118, Tsukuba, Japan, November 2005. Springer LNCS 3780.

15-819K: Logic Programming

Lecture 27

Constraint Logic Programming

Frank Pfenning

December 7, 2006

In this lecture we sketch constraint logic programming which generalizes the fixed structure of so-called uninterpreted function and predicate symbols of Horn logic. A common application is more flexible logic programming with arithmetic and finite domains. Higher-order logic programming is another example where techniques from constraint logic programming are important.

27.1 Arithmetic

One of the main motivations for constraint logic programming comes from the awkward and non-logical treatment of arithmetic in Prolog. For example, a naive implementation of the Fibonacci function would be the following Prolog program.

```
fib(0,1).
fib(1,1).
fib(N,F) :- N >= 2,
    N1 is N-1, fib(N1,F1),
    N2 is N-2, fib(N2,F2),
    F is F1+F2.
```

Recall the use of `is/2` to carry out evaluation of arithmetic predicates and the built-in `>=/2` to implement comparison between two ground terms.

Constraint logic programming supports interpreted function symbols (here: addition and subtraction) as term constructors, which means that we need to generalize unification to take into account the laws of arithmetic. Moreover, built-in predicates (here: comparison) over the constraint

domain (here: integers) are no longer restricted to ground terms but are treated specially as part of the constraint domain.

In a constraint logic programming language over the integers, we could rewrite the above program as

```
fib(0,1).
fib(1,1).
fib(N,F1+F2) :- N >= 2, fib(N-1,F1), fib(N-2,F2).
```

With respect to this program, a simple query

```
?- fib(2,F-1).
```

is perfectly legitimate and should yield `F = 1`, but even more complex queries such as

```
?- N < 20, fib(N,5).
```

and

```
?- N < 20, fib(N,6).
```

will succeed (in first case, with `N = 5`) or fail finitely (in the second case).

What emerges from the examples is that we need to extend ordinary unification to handle more general equations, with terms from the constraint domain, and that we furthermore need to generalize from just equalities to maintain other constraints such as inequalities.

27.2 An Operational Semantics with Constraints

Before we generalize to other domains, we return to the usual domain of first-order terms and reformulate proof search. The idea is to replace unification by equality constraints.

We use the residuated form of programs in order to isolate the various choices and appeals to unification. Recall the language of goals G, goal stacks S and failure continuations F. We only consider the Horn fragment, so the program is fixed. Moreoever, we assume there is exactly one residuated program clause D_p for every predicate p.

Goals	G	$::=$	$P \mid G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid \exists x.\, G \mid s \doteq t$
Programs	D	$::=$	$\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G$
Goal Stacks	S	$::=$	$\top \mid G \wedge S$
Failure Conts.	F	$::=$	$\bot \mid (G \wedge S) \vee F$

The operational semantics is given by three judgments.

- $G \;/\; S \;/\; F$. Solve G under goal stack S with failure continuation F.
- $s \doteq t \mid \theta$. Unification of s and t yields most general unifier θ.
- $s \doteq t \not\mid$. Terms s and t are not unifiable.

We now add to this a constraint store C, for now just consisting of equations.

$$\text{Constraints} \quad C \quad ::= \quad \top \mid s \doteq t \wedge C$$

The first new judgment is

- $G \;/\; S \;/\; C \;/\; F$. Solve G under goal stack S and constraint C with failure continuation F.

First, the rules for conjunction, which are not affected by the constraint except that they carry them along.

$$\frac{G_1 \;/\; G_2 \wedge S \;/\; C \;/\; F}{G_1 \wedge G_2 \;/\; S \;/\; C \;/\; F} \qquad \frac{G_2 \;/\; S \;/\; C \;/\; F}{\top \;/\; G_2 \wedge S \;/\; C \;/\; F} \qquad \frac{}{\top \;/\; \top \;/\; C \;/\; F}$$

We can see in the rule for final success that the constraint C represents a form of the answer. In practice, we project the constraints down to the variables occurring in the original query, although we do not discuss the details of the projection operation in this lecture.

For disjunction we have to remember the constraint as well as the success continuation.

$$\frac{G_1 \;/\; S \;/\; C \;/\; (G_2 \wedge S \wedge C) \vee F}{G_1 \vee G_2 \;/\; S \;/\; C \;/\; F} \qquad \frac{G_2 \;/\; S \;/\; C \;/\; F}{\bot \;/\; S' \;/\; C' \;/\; (G_2 \wedge S \wedge C) \vee F}$$

$$\begin{array}{c}\text{no rule for} \\ \bot \;/\; S \;/\; C \;/\; \bot\end{array}$$

Predicate calls in residuated form do not involve unification, so they remain unchanged from the unification-based semantics. Existential quantification is also straightforward.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G) \in \Gamma \quad G(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; C \;/\; F}{p(\mathbf{t}) \;/\; S \;/\; C \;/\; F}$$

$$\frac{G(X/x) \;/\; S \;/\; C \;/\; F \quad X \notin \mathrm{FV}(S, C, F)}{\exists x.\, G \;/\; S \;/\; C \;/\; F}$$

For equations, we no longer want to appeal to unification. Instead, we check if the new equation $s \doteq t$ together with the ones already present in C are still consistent. If so, we add the new constraint $s \doteq t$; if not we fail and backtrack.

$$\frac{s \doteq t \wedge C \not\vdash \bot \quad \top \;/\; S \;/\; s \doteq t \wedge C \;/\; F}{s \doteq t \;/\; S \;/\; C \;/\; F} \qquad \frac{s \doteq t \wedge C \vdash \bot \quad \bot \;/\; S \;/\; C \;/\; F}{s \doteq t \;/\; S \;/\; C \;/\; F}$$

We use a new judgment form, $C \vdash \bot$, to check if a set of constraints is consistent. It can be implemented simply by the left rules for equality, or by the forward chaining rules for unification described in an earlier lecture. The interpretation of variables, however, is a bit peculiar. The variables in a constraint C as part of the $G \;/\; S \;/\; C \;/\; F$ are (implicitly) existentially quantified. When we ask if the constraints are inconsistent we mean to check that $\neg\exists\mathbf{X}.\, C$, that is, there does not exist a substitution $\mathbf{t}/\mathbf{X}$ which makes $C(\mathbf{t}/\mathbf{X})$ true. We check this by assuming $\exists\mathbf{X}.\, C$ and trying to derive a contradiction. This means we introduce a new *parameter* x for each logic variable X and actually try to prove $C(\mathbf{x}/\mathbf{X})$ where each of the variables $\mathbf{x}$ is fresh. Since we are in the Horn fragment, we omitted this extra step of back-substituting parameters since there is only one kind of variable.

An interesting point about the semantics above is that we no longer use or need substitutions θ. Whenever a new equation arrives we make sure the totality of all equations encountered so far still has a solution and continue.

27.3 An Alternative Operational Semantics with Constraints

As noted, the treatment of variables in the above semantics is somewhat odd. We introduce them as logic variables, convert them to parameters to check consistency. But we never use them for anything else, so why introduce them as logic variables in the first place? Another jarring aspect of the semantics is that the work that goes into determining that the equations are consistent (for example, with the left rules for unifiability from an earlier lecture) is lost after the check, and we may have to redo a good bit of work when the next equality is encountered. In other words, the constraints are not solved *incrementally*.

This suggests the following change in perspective: rather than trying to prove that there *exists* a unifying substitution, we think of search as trying to characterize *all* unifying substitutions. We still need to treat the case that there are none as special (so we can fail), but otherwise we just *assume* constraints. Reverting back to pure logic for a moment, a sequent $C \vdash A$

with parameters $\mathbf{x}$ holds if any substitution $\mathbf{t}/\mathbf{x}$ which makes C true also makes A true.

Once constraints appear on the left-hand side, they can be treated with the usual left rules. The main judgment is now $\mathcal{C} \vdash G \;/\; S \;/\; F$ for a set of constraints $\mathcal{C}$ where we maintain the invariant that $\mathcal{C}$ is always satisfiable (that is, it is never the case that $\mathcal{C} \vdash \bot$). This should be parenthesized as $(\mathcal{C} \vdash G \;/\; S) \;/\; F$ because the constraints $\mathcal{C}$ do not apply to F.

For most of the rules from above this is a mere notational change. We a few interesting cases. We generalize the left-hand side slightly to be a collection of constraints $\mathcal{C}$ instead of a single one.

$$\frac{\mathcal{C} \vdash G_1 \;/\; S \;/\; (\mathcal{C} \vdash G_2 \wedge S) \vee F}{\mathcal{C} \vdash G_1 \vee G_2 \;/\; S \;/\; F} \qquad \frac{\mathcal{C} \vdash G_2 \;/\; S \;/\; F}{\mathcal{C}' \vdash \bot \;/\; S' \;/\; (\mathcal{C} \vdash G_2 \wedge S) \vee F}$$

no rule for
$\mathcal{C} \vdash \bot \;/\; S \;/\; \bot$

Existential quantification now introduces a new parameter.

$$\frac{\mathcal{C} \vdash G \;/\; S \;/\; F \quad x \notin \mathrm{FV}(S, \mathcal{C}, F)}{\mathcal{C} \vdash \exists x.\, G \;/\; S \;/\; F}$$

We avoid the issue of types and a typed context of parameters as we discussed in the lecture of parameters.

Equality is now treated differently.

$$\frac{\mathcal{C}, s \doteq t \not\vdash \bot \quad \mathcal{C}, s \doteq t \vdash \top \;/\; S \;/\; F}{\mathcal{C} \vdash s \doteq t \;/\; S \;/\; F} \qquad \frac{\mathcal{C}, s \doteq t \vdash \bot \quad \mathcal{C} \vdash \bot \;/\; S \;/\; F}{\mathcal{C} \vdash s \doteq t \;/\; S \;/\; F}$$

Now there is scope for various left rules concerning equality. The simplest example is the left rule for equality discussed in an earlier lecture. This actually recovers the usual unification semantics!

$$\frac{s \doteq t \mid \theta \quad (\mathcal{C}\theta \vdash G\theta \;/\; S\theta) \;/\; F}{(\mathcal{C}, s \doteq t \vdash G \;/\; S) \;/\; F}$$

Note that the other case of the left rule (where s and t do not have a unifier) cannot arise because of our satisfiability invariant which guarantees that a unifier exists.

We can also use the small-step rules dealing with equality that will never apply a substitution, just accumulate information about the variables. For example, knowing $x \doteq c$ for a constant c carries the same information as applying the substitution c/x.

These left rules will put a satisfiable constraint into a kind of reduced form and in practice this is combined with the satisfiability check. This means constraints are treated incrementally, which is of great practical importance especially in complex constraint domains.

As a final remark, we come back to focusing. The rules for equality create a kind of non-determinism, because either we could solve a goal or we could break down the equality we just assumed. However, the rules for equality are asynchronous on the left and can be reduced eagerly until we get irreducible equations. In a complete, lower-level semantics this should be addressed explicitly; we omit this step here and leave it as Exercise 27.1.

27.4 Richer Constraint Domains

The generalization to richer domains is now not difficult. Instead of just equalities, the constraint C (or the constraint collection $\mathcal{C}$) contains other interpreted predicate symbols such as inequalities of even disequalities. When encountering an equality or interpreted predicate we verify its consistency, adding it to the set of constraints.

In addition we allow either constraint simplification, or saturate left rules for the predicates in the constraint domain. The simplification algorithms depend significantly on the particular constraint domains. For example, for arithmetic equalities we might use Gaussian elimination, for arithmetic inequalities the simplex algorithm. In addition we need to consider combinations of constraint domains, for which there are general architectures such as the Nelson-Oppen method for combining decision procedures.

A particularly popular constraint domain is Finite Domains (FD), which is supported in implementations such as GNU Prolog. This also supports bounded arithmetic as a special case. We will not go into further detail, except to say that the Fibonacci example is expressible in several constraint languages.

27.5 Hard Constraints

An important concept in practical constraint domains is that of a *hard constraint*. Hard constraints may be difficult to solve, or may even be undecidable. The general strategy in constraint programming language is to postpone the solution of hard constraints until further instantiations make them tractable. An example might be

```
?- X * Y = 4, X = 2.
```

When we see `X * Y = 4`, the equation is non-linear, so we would be justified in raising an exception if the domain was supposed to treat only linear equations. But when we receive the second constraint, `X = 2`, we can simplify the first constraint to be linear `2 * Y = 4` and simplify to `Y = 2`.

When hard constraints are left after overall "success", the success must be interpreted conditionally: any solution to the remaining hard constraints yields a solution to the overall query. It is even possible that the hard constraints may have no solution, negating an apparent success, so extra care must be taken when the interpreter admits hard constraints.

Hard constraints arise naturally in arithmetic. Another domain where hard constraints play a significant role is that of terms containing abstractions (*higher-order abstract syntax*), where constraint solving is a form of higher-order unification. This is employed, for example, in the Twelf systems, where hard constraints (those falling outside the pattern fragment) are postponed and reawakened when more information may make them tractable.

27.6 Detailed Example

As our example we consider the Fibonacci sequence again.

```
fib(0,1).
fib(1,1).
fib(N,F1+F2) :- N >= 2, fib(N-1,F1), fib(N-2,F2).
```

We use it in this direct form, rather then the residuated form for brevity. We consider the query

```
?- N < 10, fib(N,2).
```

which inverts the Fibonacci functions, asking for which $n < 10$ we have $\mathsf{fib}(n) = 2$. The bound on n is to avoid possible non-termination, although here it would only affect search after the first solution. Inverting the Fibonacci function directly as with this query is impossible with ordinary Prolog programs.

Below we show $G \mid S \mid C$, omitting the failure continuation and silently simplifying constraints on occasion. We avoid redundant "$\wedge \top$" and use Prolog notation throughout. Furthermore, we have substituted for the first occurrence of a variable in a clause head instead of building an equality constraint.

```
N < 10, fibr(N,2) / true / true
fib(N,2) / true / N < 10
% trying clause fib(0,1)
% 0 = N , 1 = 2, N < 10  is inconsistent
% trying clause fib(1,1)
% 1 = N , 1 = 2, N < 10  is inconsistent
% trying clause fib(N,F1+F2) :- ...
F1+F2 = 2 / N >= 2, fib(N-1,F1), fib(N-2,F2) / N < 10
N >= 2 / fib(N-1,F1), fib(N-2,F2) / F1 = 2-F2, N < 10
fib(N-1,F1) / fib(N-2,F2) / F1 = 2-F2, 2 <= N, N < 10
% trying clause fib(0,1)
% 0 = N-1, 1 = F1, F1 = 2-F2, 2 <= N, N < 10  is incons.
% trying clause fib(1,1)
1 = N-1, 1 = F1 / fib(N-2,F2) / F1 = 2-F2, 2 <= N, N < 10
fib(N-2,F2) / true / F1 = 2-F2, N = 2
% trying clause fib(0,1)
0 = N-2, 1 = F2 / true / F1 = 2-F2, N = 2
true / true / 0 = N-2, 1 = F2, F1 = 2-F2, N = 2
true / true / N = 2, F2 = 1, F1 = 1
```

Even though GNU Prolog offers finite domain constraints, including integer ranges, the Fibonacci program above does not quite run as given. The problem is that, in order to be backward compatible with Prolog, the predicates of the constraint domain (including equality) must be separated out. The naming convention is to precede a predicate with `#` to obtain the corresponding constraint predicate (assuming it is defined). Here is a bi-directional version of the Fibonacci predicate in GNU Prolog.

```
fibc(0,1).
fibc(1,1).
fibc(N,F) :- N #>= 2,
    N1 #= N-1, fibc(N1,F1),
    N2 #= N-2, fibc(N2,F2),
    F #= F1+F2.
```

With this predicate we can execute queries such as

```
?- N #< 10, fibc(N,8).
```

(which succeeds) and

```
?- N #< 10, fibc(N,9).
```

(which fails). A query

```
?- N < 10, fibc(N,8).
```

would signal an error, because the first argument to < is not ground.

27.7 Historical Notes

Constraint logic programming was first proposed by Jaffar and Lassez [3]. The first practical implementation was by Jaffar and Michaylov [4], the full CLP(R) language and system later described by Jaffar et al. [5]. A related language is Prolog III [1] which combines several constraint domains. The view of higher-order logic programming as constraint logic programming was advanced by Michaylov and myself [8, 6].

The architecture of cooperating decision procedures is due to Nelson and Oppen [7].

In the above constraint logic programming language the constraints and their solution algorithms are hard-wired into the language. The sublanguage of Constraint Handling Rules (CHR) [2] aims at allowing the specification of constraint simplification within the language for greater flexibility. It seems that this is a fragment of LolliMon, specifically, its linear forward chaining sublanguage, which could be the basis for a more logical explanation of constraints and constraint simplification in logic programming.

27.8 Exercises

Exercise 27.1 *Write a semantics for Horn logic where unification is replaced by incremental constraint solving as sketched in this lecture. Make sure your rules have no unwanted non-determinism, that is, they can be viewed as a deterministic abstract machine.*

27.9 References

[1] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[2] Thom Früwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 17(1–3):95–138, October 1998.

[3] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[4] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming (ICLP'87)*, pages 196–218, Melbourne, Australia, May 1987. MIT Press.

[5] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[6] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[7] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[8] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.