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.

$$\frac{s \doteq t \wedge C \nvdash \bot \quad \top \;/\; S \;/\; s \doteq t \wedge C \;/\; F}{s \doteq t \;/\; S \;/\; C \;/\; F} \qquad \frac{s \doteq t \wedge C \vdash \bot \quad \bot \;/\; S \;/\; C \;/\; F}{s \doteq t \;/\; S \;/\; C \;/\; F}$$

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.

$$\frac{\mathcal{C} \vdash G_1 \;/\; S \;/\; (\mathcal{C} \vdash G_2 \wedge S) \vee F}{\mathcal{C} \vdash G_1 \vee G_2 \;/\; S \;/\; F} \qquad \frac{\mathcal{C} \vdash G_2 \;/\; S \;/\; F}{\mathcal{C}' \vdash \bot \;/\; S' \;/\; (\mathcal{C} \vdash G_2 \wedge S) \vee F}$$

$$\begin{array}{c}\text{no rule for}\\ \mathcal{C} \vdash \bot \;/\; S \;/\; \bot\end{array}$$

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.

$$\frac{\mathcal{C}, s \doteq t \nvdash \bot \quad \mathcal{C}, s \doteq t \vdash \top \;/\; S \;/\; F}{\mathcal{C} \vdash s \doteq t \;/\; S \;/\; F} \qquad \frac{\mathcal{C}, s \doteq t \vdash \bot \quad \mathcal{C} \vdash \bot \;/\; S \;/\; F}{\mathcal{C} \vdash s \doteq t \;/\; S \;/\; F}$$

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 $G \ / \ S \ / \ C$, 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.