Constructive Negation in Logic Programming

Roman Barták
E-mail: bartak@ktisun1.ms.mff.cuni.cz
Department of Computer Science
Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic

logic programs = definite clause programs = no negation

including negation $\Rightarrow$ superexponencial time complexity

History:

- Closed World Assumption (Reiter-databases)
- ✓ Negation as Failure (Clark)
- Generalized Logic Programs (Clark)
- ✓ Constructive Negation (Chan)

Negation as Failure

- introduced by K.J. Clark in 1978
- better: negation as finite failure
- not logical negation
- imperative character

```
not P:-P,!,fail.
not P.
```

- most common in PROLOG programming
- attractive from a programming point of view
  - – exploits the efficiency of the underlying program
  - – handles "special" features of the language, e.g. cut
- two important drawbacks
  - – can only handle ground negative subgoals
  - – can only be used as a test

    cannot generate any new bindings for query variables
- non-intuitive results

Program	Goals	Results
u(a). u(b).	not u(X),v(X)	no
v(a). v(c).	v(X),not u(X)	X=c

Introduction to Constructive Negation

- introduced by D. Chan in 1988
- *constructive negation* = an extension of the negation as failure to handle non-ground negative subgoal in a constructive manner
- involves:
  - - taking a negative goal
  - - running the positive version
  - - negating the collected answers (the key part)

    introduces quantified disequations $\forall Z\ s \neq t$
- inherits many of the advantages of negation as failure
  - – exploits the efficiency of the underlying program
  - – handles "special" features of the language, e.g. cut
- removes drawbacks
  - – can handle non-ground negative subgoals
  - – can generate new bindings for query variables
- constructive negation for Constraint Logic Programming

  Stuckey 1991

  more natural framework

Handling Equalities

- correspond to substitutions
- types of equalities:

  $t=u$ is *valid* iff $\forall\sigma\ t\sigma \equiv u\sigma$ $\qquad f(a,Y) = f(a,Y)$

  $t=u$ is *invalid* iff $\forall\sigma\ \neg(t\sigma \equiv u\sigma)$ $\qquad f(X,g(a)) = f(X,b)$

  $t=u$ is *satisfiable* iff $\exists\sigma\ t\sigma \equiv u\sigma$ $\qquad f(X) = f(Y)$
- *normal form* of equalities

  conjunction (set) of equalities $x_i = t_i$, where x_i are variables & $\forall i,j\ x_i \notin vars(t_j)$

  corresponds to the most general unifier
- two sets of equalities are *equivalent* if they have the same unifiers
- Theorems:
  - ⋆ The type of equality (valid/satisfiable/invalid) is algorithmic decidable.
  - ⋆ Every satisfiable equality is transferable (by program) to equivalent normal form. (Unification Theorem)

Handling Disequalities
(Definitions)

- types of disequalities:

  $t \neq u$ is *valid* iff $\forall\sigma\ \neg(t\sigma \equiv u\sigma)$ $\qquad f(X,g(a)) \neq f(X,b)$

  $t \neq u$ is *invalid* iff $\forall\sigma\ t\sigma \equiv u\sigma$ $\qquad f(a,Y) \neq f(a,Y)$

  $t \neq u$ is *satisfiable* iff $\exists\sigma\ \neg(t\sigma \equiv u\sigma)$ $\qquad f(X) \neq f(Y)$

- $t_1 \neq u_1$ and $t_2 \neq u_2$ are *similar* $(t_1 \neq u_1 \approx t_2 \neq u_2)$
  iff $(t_1 \equiv t_2 \ \&\ u_1 \equiv u_2) \vee (t_1 \equiv u_2 \ \&\ u_1 \equiv t_2)$
- normal forms of disequalities

  simple unequality (SUE) $x \neq t$
  iff x is variable & $x \notin vars(t)$ $\qquad \ldots X \neq f(Y)$

  complex unequality (CUE) $[x_i] \neq [t_i]$
  iff $x_i \neq t_i$ is SUE ($SUE_i \in CUE$) $i=1\ldots n$, $n>1$ &
  $\forall i \neq j\ \neg(SUE_i \approx SUE_j)$ $\qquad \ldots [X,Y] \neq [a,g(Z)]$
- $t_1 \neq u_1$ and $t_2 \neq u_2$ are *equivalent* $(t_1 \neq u_1 \leftrightarrow t_2 \neq u_2)$
  iff $\forall\sigma\ (t_1\sigma \equiv u_1\sigma) \Leftrightarrow (t_2\sigma \equiv u_2\sigma)$

to be continued

Handling Disequalities
(Theorems)

- Corollary:
  Similar disequalities are equivalent.
- Theorems:
  - ⋆ The type of unequality (valid/satisfiable/invalid) is algorithmic decidable.
  - ⋆ Every satisfiable but not valid unequality is transferable (by program) to equivalent normal form (SUE or CUE).
- conjunction of disequalities

  covering relationship ($\angle$)

  $SUE \angle SUE'$ iff $SUE \approx SUE'$

  $SUE \angle CUE$ iff $\exists SUE' \in CUE\ SUE \approx SUE'$

  $CUE \angle CUE'$ iff $\forall SUE \in CUE\ \exists SUE' \in CUE'\ SUE \approx SUE'$

  examples:
  $X \neq Y \angle Y \neq X$
  $X \neq a \angle [X,Y] \neq [a,b]$
  $[X,Y] \neq [a,b] \angle [X,Y,X] \neq [a,b,Y]$

  Corollary: $U \angle V \Leftrightarrow (U \ \&\ V) \leftrightarrow U$

Solution Filter

result = set of equalities and disequalities

What should be presented to the user (to higher level)?

two extremes:

- nothing ≈ yes/no
- everything

unsuitable for constructive negation

- relevant equalities (PROLOG)
- ✓ relevant equalities and disequalities

Some definitions:

the *equality* x=t is *relevant* to the set of variables V
iff $(x \in V \vee t \in V)$

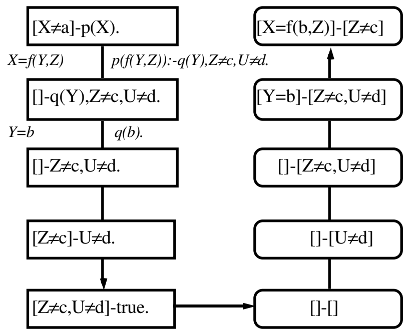
the *SUE* x≠t is *relevant* to the set of variables V
iff $(x \in V \vee t \in V)$

the *CUE* t≠u is *relevant* to the set of variables V
iff $\forall SUE_i \in CUE$ (SUE_i is relevant to V)

Example of Solution Filter

program: p(a).
p(f(Y,Z)):-q(Y),Z≠c,U≠d.
q(b).

solution process for the task X≠a & p(X):



Constructive Negation
(Some Practical Aspects)

- involves:
  - taking a negative goal
  - running the positive version and collecting answers
  - negating the collected answers (the key part)
- negation of the collected answers

  based on formula:

  $\neg\exists Y,Z\ (x=t\ \&\ Q) \Leftrightarrow \forall Y\ (x \neq t) \vee \exists Y\ (x=t\ \&\ \neg\exists Z\ Q)$
- example:

  program: p(a).
  p(f(Y)):-Y≠b.

  goal: ?-not p(X).

  positive version of the goal: ?-p(X).

  collected answers: X=a ∨ (X=f(Y) & Y≠b)

  negated answers: (X≠a & X≠f(Y)) ∨ X=f(b)

  be careful: ~~(X≠a & X≠f(Y)) ∨ (X≠a & Y≠b)~~
  ~~(X≠a & X≠f(Y)) ∨ X≠a~~

Negation as Failure vs. Constructive Negation
(Examples)

Programs	Goals	Negation as Failure	Constructive Negation
p(f(Y)):- not q(Y). q(a).	p(X)	no	X=f(Y),Y≠a
	not p(X)	yes	X≠f(Y) X=f(a)
	not not p(X)	no	X=f(Y),Y≠a
s(f(Y)):- not r(Y). r(Z).	s(X)	no	no
	not s(X)	yes	yes
p(a,f(Z)):-t(Z). p(f(Z),b):-t(Z). t(c).	not p(X,Y)	no	X≠a,X≠f(c) X≠f(c),Y≠f(c) X≠a,Y≠b Y≠f(c),Y≠b
u(a). u(b). v(a). v(c).	not(u(X),v(X))	no	X≠a
	not u(X),not v(X)	no	X≠a,X≠b,X≠c
	not u(X),v(X)	no	X=c
	v(X),not u(X)	X=c	X=c