Constraint Logic Programming

Sylvain Soliman and François Fages
{Sylvain.Soliman,Francois.Fages}@inria.fr

INRIA – Project-Team CONTRAINTES

MPRI C-2-4-1 Course – September 2009 - February 2010

Part III

CLP - Operational and Fixpoint Semantics

Part III: CLP - Operational and Fixpoint Semantics

Part IV

Logical Semantics

Part IV: Logical Semantics

Undecidability of $M_P^{\mathcal{X}}$

```
loop:- loop.
contr(P):- success(P,P), loop.
contr(P):- fail(P,P).
```

If `contr(contr)` has a success,
then `success(contr,contr)` succeeds,
and `fail(contr,contr)` doesn't succeed,
hence `contr(contr)` doesn't succeed: contradiction.

If `contr(contr)` doesn't succeed,
then `fail(contr,contr)` succeeds,
hence `contr(contr)` succeeds: contradiction.

Therefore programs `success` and `fail` cannot both exist.

Clark's completion

The Clark's completion of P is the set P^* of formulas of the form
$\forall X\ p(X) \leftrightarrow (\exists Y_1 c_1 \wedge A^1_1 \wedge \ldots \wedge A^1_{n_1}) \vee \ldots \vee (\exists Y_k c_k \wedge A^k_1 \wedge \ldots \wedge A^k_{n_k})$
where the $p(X) \leftarrow c_i | A^i_1, \ldots, A^i_{n_i}$ are the rules in P and Y_i's the local variables,
$\forall X \neg p(X)$ if p is not defined in P.

Example 1

CLP($\mathcal{H}$) program `p(s(X)):- p(X).`
Clark's completion $P^* = \{\forall x\ p(x) \leftrightarrow \exists y\ x = s(y) \wedge p(y)\}$.
The goal `p(0)` finitely fails, we have $P^*, CET \models \neg p(0)$.
The goal `p(X)` doesn't finitely fail,
we have $P^*, CET \not\models \neg\exists X\ p(X)$ although $P^* \models_{\mathcal{H}} \neg\exists X\ p(X)$

Models of the Clark's completion

Theorem 2

i) P^* has the same least $\mathcal{X}$-model than P, $M_P^{\mathcal{X}} = M_{P^*}^{\mathcal{X}}$
ii) $P \models_{\mathcal{X}} c \supset A$ iff $P^* \models_{\mathcal{X}} c \supset A$, for all c and A,
iii) $P, \mathcal{T} \models c \supset A$ iff $P^*, \mathcal{T} \models c \supset A$.

Proof.

i) is an immediate corollary of full abstraction and least $\mathcal{X}$-model theorems
For iii) we clearly have $(P, \mathcal{T} \models c \supset A) \Rightarrow (P^*, \mathcal{T} \models c \supset A)$. We show the contrapositive of the opposite, $(P, \mathcal{T} \not\models c \supset A) \Rightarrow (P^*, \mathcal{T} \not\models c \supset A)$.
Let I be a model of P and $\mathcal{T}$, based on a structure $\mathcal{X}$, let ρ be a valuation such that $I \models \neg A\rho$ and $\mathcal{X} \models c\rho$.
We have $M_P^{\mathcal{X}} \models \neg A\rho$, thus $M_{P^*}^{\mathcal{X}} \models \neg A\rho$, and as $\mathcal{T} \models c\rho$, we conclude that $P^*, \mathcal{T} \not\models c \supset A$.
The proof of ii) is identical, the structure $\mathcal{X}$ being fixed. □

Soundness of Negation as Finite Failure

Theorem 3

If G is finitely failed then $P^*, \mathcal{T} \models \neg G$.

Proof.

By induction on the height h of the tree in finite failure for $G = c|A, \alpha$ where A is the selected atom at the root of the tree.
In the base case $h = 1$, the constrained atom $c|A$ has no CSLD transition, we can deduce that $P^*, \mathcal{T} \models \neg(c \wedge A)$ hence that $P^*, \mathcal{T} \models \neg G$.
For the induction step, let us suppose $h > 1$. Let $G_1, \ldots, G_n$ be the sons of the root and $Y_1, \ldots, Y_n$ be the respective sets of introduced variables.
We have $P^*, \mathcal{T} \models G \leftrightarrow \exists Y_1\ G_1 \vee \ldots \vee \exists_n\ G_n$. By induction hypothesis, $P^*, \mathcal{T} \models \neg G_i$ for every $1 \leq i \leq n$, therefore $P^*, \mathcal{T} \models \neg G$. □

INRIA

Completeness of Negation as Failure

Theorem 4 ([JL87])

If $P^*, \mathcal{T} \models \neg G$ then G is finitely failed.

We show that if G is not finitely failed then $P^*, \mathcal{T}, \exists(G)$ is satisfiable. If G has a success then by the soundness of CSLD resolution, $P^*, \mathcal{T} \models \exists G$. Else G has a fair infinite derivation $G = c_0|G_0 \longrightarrow c_1|G1 \longrightarrow \ldots$
For every $i \geq 0$, c_i is $\mathcal{T}$-satisfiable, thus by the compactness theorem, $c_\omega = \bigwedge_{i \geq 0} c_i$ is $\mathcal{T}$-satisfiable. Let $\mathcal{X}$ be a model of $\mathcal{T}$ s.t. $\mathcal{X} \models \exists(c_\omega)$. Let $I_0 = \{A\rho \mid A \in G_i$ for some $i \geq 0$ and $\mathcal{X} \models c_\omega \rho\}$. As the derivation is fair, every atom A in I_0 is selected, thus $c_\omega|A \longrightarrow c_\omega|A_1, \ldots, A_n$ with $[c_\omega|A] \cup \ldots \cup [c_\omega|A_n] \subseteq I_0$. We deduce that $I_0 \subseteq T_P^{\mathcal{X}}(I_0)$. By Knaster-Tarski's theorem, the iterated application up to ordinal ω of the operator $T_P^{\mathcal{X}}$ from I_0 leads to a fixed point I s.t. $I_0 \subseteq I$, thus $[c_\omega|G_0] \in I$. Hence $P^*, \exists(G)$ is $\mathcal{X}$-satisfiable, and $P^*, \mathcal{T}, \exists(G)$ is satisfiable.

Part V

Practical CLP Programming

The Warren Abstract Machine

First Prolog implementation in the early 70's (by Colmerauer et al.).

In 1983, David H. Warren creates the Warren Abstract Machine.

Remains the state of the art (for term representation, basic instructions, . . .)

Slightly extended for CLP

(C)SLD resolution seen as a call stack (with marks for choice points)

The Warren Abstract Machine

First Prolog implementation in the early 70's (by Colmerauer et al.).

In 1983, David H. Warren creates the Warren Abstract Machine.

Remains the state of the art (for term representation, basic instructions, . . .)

Slightly extended for CLP (constraints instead of substitutions)

(C)SLD resolution seen as a call stack (with marks for choice points)

Optimizations from the WAM

Search for predicates should be almost in constant time

Use a hash table - indexing - for the predicate name/arity,

Each call normally adds a frame to the call stack (removed on backtracking)

As for other programming paradigms, not always necessary

Optimizations from the WAM

Search for predicates should be almost in constant time

Use a hash table - indexing - for the predicate name/arity, and the functor of the first argument

Each call normally adds a frame to the call stack (removed on backtracking)

As for other programming paradigms, not always necessary

Optimizations from the WAM

Search for predicates should be almost in constant time

Use a hash table - indexing - for the predicate name/arity, and the functor of the first argument

Each call normally adds a frame to the call stack (removed on backtracking)

As for other programming paradigms, not always necessary

Tail recursion can be optimized,

Optimizations from the WAM

Search for predicates should be almost in constant time

Use a hash table - indexing - for the predicate name/arity, and the functor of the first argument

Each call normally adds a frame to the call stack (removed on backtracking)

As for other programming paradigms, not always necessary

Tail recursion can be optimized, when calling and called contexts are deterministic.

Putting it all together

Naive sum

```
sum([], 0).
sum([H | T], S) :-
   sum(T, S1),
   S is S1 + H.
```

Putting it all together

Naive sum

```
sum([], 0).
sum([H | T], S) :-
   sum(T, S1),
   S is S1 + H.
```

Much better

```
sum(L, S) :-
   sum_aux(L, 0, S).

sum_aux([], S, S).
sum_aux([H | T], S0, S) :-
   S1 is S0 + H,
   sum_aux(T, S1, S).
```

Putting it all together

If numbers are coded as the fact `number(X)`?

```
sum(S) :- findall(X, number(X), L), sum(L, S).
```

Putting it all together

If numbers are coded as the fact number(X)?

```
sum(S) :- findall(X, number(X), L), sum(L, S).
```

```
sum(S) :-
   g_assign(sum, 0),
   (
      number(N),
      g_read(sum, S1),
      S2 is S1 + N,
      g_assign(sum, S2),
      fail
   ;
      g_read(sum, S)
   ).
```

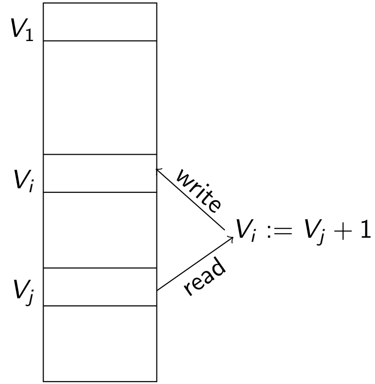
Part VI

Concurrent Constraint Programming

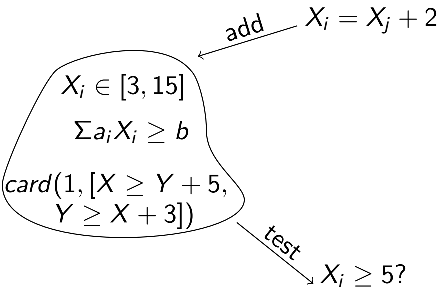
Part VI: Concurrent Constraint Programming

The Paradigm of Constraint Programming

memory of values
programming variables



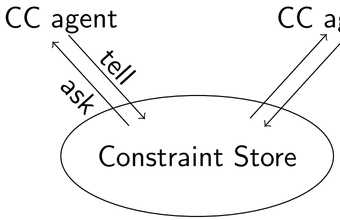
memory of constraints
mathematical variables

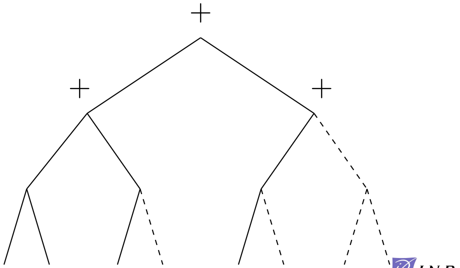


Concurrent Constraint Programs

Class of programming languages CC($\mathcal{X}$) introduced by Saraswat [Sar93] as a merge of Constraint and Concurrent Logic Programming.

Processes	$P ::= \mathcal{D}.A$
Declarations	$\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$
Agents	$A ::= tell(c) \mid \qquad \mid A \parallel A \mid A + A \mid \exists x A \mid p(\vec{x})$

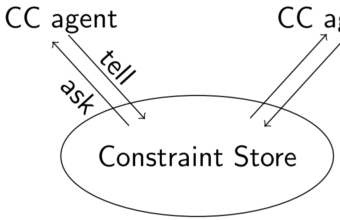


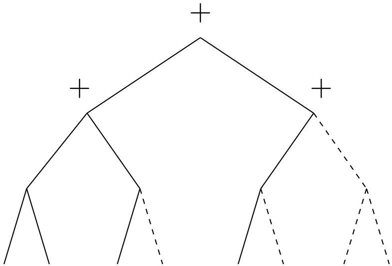


Concurrent Constraint Programs

Class of programming languages CC($\mathcal{X}$) introduced by Saraswat [Sar93] as a merge of Constraint and Concurrent Logic Programming.

Processes	$P ::= \mathcal{D}.A$
Declarations	$\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$
Agents	$A ::= tell(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid A + A \mid \exists x A \mid p(\vec{x})$





Translating CLP($\mathcal{X}$) into CC($\mathcal{X}$) Declarations

CLP($\mathcal{X}$) program:

$$A \leftarrow c|B, C$$
$$A \leftarrow d|D, E$$
$$B \leftarrow e$$

equivalent CC($\mathcal{X}$) declaration:

$$A = tell(c)||B||C + tell(d)||D||E$$
$$B = tell(e)$$

This is just a process calculus syntax for CLP programs. . .

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

$$(\text{CC agent})^\dagger = \text{CLP goal}$$

$$(tell(c))^\dagger =$$

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

$$(\text{CC agent})^\dagger = \text{CLP goal}$$

$$\begin{aligned} (tell(c))^\dagger &= c \\ (A \mid\mid B)^\dagger &= \end{aligned}$$

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

$$(\text{CC agent})^\dagger = \text{CLP goal}$$

$$\begin{aligned}
(tell(c))^\dagger &= c\\
(A \parallel B)^\dagger &= A^\dagger, B^\dagger\\
(A + B)^\dagger &=
\end{aligned}$$

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

$$(\text{CC agent})^\dagger = \text{CLP goal}$$

$$
\begin{aligned}
(tell(c))^\dagger &= c \\
(A \mid\mid B)^\dagger &= A^\dagger, B^\dagger \\
(A + B)^\dagger &= p(\vec{x}) \text{ where } \vec{x} = fv(A) \cup fv(B) \text{ and} \\
&\qquad p(\vec{x}) \leftarrow A^\dagger \\
&\qquad p(\vec{x}) \leftarrow B^\dagger \\
(\exists x\ A)^\dagger &=
\end{aligned}
$$

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

$$(\text{CC agent})^\dagger = \text{CLP goal}$$

$$\begin{aligned}
(tell(c))^\dagger &= c\\
(A \mid\mid B)^\dagger &= A^\dagger, B^\dagger\\
(A + B)^\dagger &= p(\vec{x}) \text{ where } \vec{x} = fv(A) \cup fv(B) \text{ and}\\
&\qquad p(\vec{x}) \leftarrow A^\dagger\\
&\qquad p(\vec{x}) \leftarrow B^\dagger\\
(\exists x\ A)^\dagger &= q(\vec{y}) \text{ where } \vec{y} = fv(A) \setminus \{x\} \text{ and}\\
&\qquad q(\vec{y}) \leftarrow A^\dagger\\
(p(\vec{x}))^\dagger &=
\end{aligned}$$

Translating CC($\mathcal{X}$) without ask into CLP($\mathcal{X}$)

(CC agent)† = CLP goal

$$
\begin{aligned}
(tell(c))^\dagger &= c \\
(A \mid\mid B)^\dagger &= A^\dagger, B^\dagger \\
(A + B)^\dagger &= p(\vec{x}) \text{ where } \vec{x} = fv(A) \cup fv(B) \text{ and} \\
&\qquad p(\vec{x}) \leftarrow A^\dagger \\
&\qquad p(\vec{x}) \leftarrow B^\dagger \\
(\exists x\ A)^\dagger &= q(\vec{y}) \text{ where } \vec{y} = fv(A) \setminus \{x\} \text{ and} \\
&\qquad q(\vec{y}) \leftarrow A^\dagger \\
(p(\vec{x}))^\dagger &= p(\vec{x})
\end{aligned}
$$

The ask operation $c \rightarrow A$ has no CLP equivalent.

It is a new synchronization primitive between agents.

CC Computations

$$\text{Concurrency} = \text{communication (shared variables)} + \text{synchronization (ask)}$$

Communication channels, i.e. variables, are transmissible by agents (like in π-calculus, unlike CCS, CSP, Occam,...)

Communication is additive (a constraint will never be removed), monotonic accumulation of information in the store (as in CLP, as in Scott's information systems)

Synchronization makes computation both data-driven and goal-directed.

No private communication, all agents sharing a variable will see a constraint posted on that variable,

Not a parallel implementation model.

CC($\mathcal{X}$) Configurations

Configuration $(\vec{x}; c; \Gamma)$: store c of constraints, multiset Γ of agents, modulo $\equiv$ the smallest congruence s.t.:

$\mathcal{X}$-equivalence $$\frac{c \dashv\vdash_{\mathcal{X}} d}{c \equiv d}$$

α-Conversion $$\frac{z \notin fv(A)}{\exists y A \equiv \exists z A[z/y]}$$

Parallel $$(\vec{x}; c; A \parallel B, \Gamma) \equiv (\vec{x}; c; A, B, \Gamma)$$

Hiding $$\frac{y \notin fv(c,\Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)} \quad \frac{y \notin fv(c,\Gamma)}{(\vec{x}, y; c; \Gamma) \equiv (\vec{x}; c; \Gamma)}$$

CC($\mathcal{X}$) Transitions

Interleaving semantics

Procedure call $$\frac{(p(\vec{y}) = A) \in \mathcal{D}}{(\vec{x}; c; p(\vec{y}), \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)}$$

Tell $$(\vec{x}; c; tell(d), \Gamma) \longrightarrow (\vec{x}; c \wedge d; \Gamma)$$

Ask

Blind choice (local/internal)
$$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)$$
$$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; B, \Gamma)$$

CC($\mathcal{X}$) Transitions

Interleaving semantics

Procedure call

$$\frac{(p(\vec{y}) = A) \in \mathcal{D}}{(\vec{x}; c; p(\vec{y}), \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)}$$

Tell

$$(\vec{x}; c; tell(d), \Gamma) \longrightarrow (\vec{x}; c \wedge d; \Gamma)$$

Ask

$$\frac{c \vdash_{\mathcal{X}} d[\vec{t}/\vec{y}]}{(\vec{x}; c; \forall \vec{y}(d \rightarrow A), \Gamma) \longrightarrow (\vec{x}; c; A[\vec{t}/\vec{y}], \Gamma)}$$

Blind choice (local/internal)

$$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)$$
$$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; B, \Gamma)$$

$CC(\mathcal{X})$ extra rules

Guarded choice (global/external)

$$\frac{c \vdash_{\mathcal{X}} c_j}{(\vec{x}; c; \Sigma_i c_i \rightarrow A_i, \Gamma) \longrightarrow (\vec{x}; c; A_j, \Gamma)}$$

AskNot

$$\frac{c \vdash_{\mathcal{X}} \neg d}{(\vec{x}; c; \forall \vec{y}(d \rightarrow A), \Gamma) \longrightarrow (\vec{x}; c; \Gamma)}$$

Sequentiality

$$\frac{(\vec{x}; c; \Gamma) \longrightarrow (\vec{x}; d; \Gamma')}{(\vec{x}; c; (\Gamma; \Delta), \Phi) \longrightarrow (\vec{x}; d; (\Gamma'; \Delta), \Phi)}$$

$$(\vec{x}; c; (\emptyset; \Gamma), \Delta) \longrightarrow (\vec{x}; d; \Gamma, \Delta)$$

Properties of CC Transitions (1)

Theorem 5 (Monotonicity)

If $(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)$ *then* $(\vec{x}; c \wedge e; \Gamma, \Sigma) \longrightarrow (\vec{y}; d \wedge e; \Delta, \Sigma)$ *for every constraint* e *and agents* Σ*.*

Proof.

□

Corollary 6

Strong fairness and weak fairness are equivalent.

Properties of CC Transitions (1)

Theorem 5 (Monotonicity)

If $(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)$ *then* $(\vec{x}; c \wedge e; \Gamma, \Sigma) \longrightarrow (\vec{y}; d \wedge e; \Delta, \Sigma)$ *for every constraint* e *and agents* Σ.

Proof.

tell and *ask* are monotonic (monotonic conditions in guards). □

Corollary 6

Strong fairness and weak fairness are equivalent.

Properties of CC Transitions (2)

A configuration without $+$ is called deterministic.

Theorem 7 (Confluence)

For any deterministic *configuration* κ *with* deterministic *declarations,*
if $\kappa \longrightarrow \kappa_1$ *and* $\kappa \longrightarrow \kappa_2$ *then* $\kappa_1 \longrightarrow \kappa'$ *and* $\kappa_2 \longrightarrow \kappa'$ *for some* κ'.

Corollary 8

Independence of the scheduling of the execution of parallel agents.

Properties of CC Transitions (3)

Theorem 9 (Extensivity)

If $(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)$ *then* $\exists \vec{y} d \vdash_{\mathcal{X}} \exists \vec{x} c$.

Proof.

□

Theorem 10 (Restartability)

If $(\vec{x}; c; \Gamma) \longrightarrow^* (\vec{y}; d; \Delta)$ *then* $(\vec{x}; \exists \vec{y} d; \Gamma) \longrightarrow^* (\vec{y}; d; \Delta)$.

Proof.

By extensivity and monotonicity. □

Properties of CC Transitions (3)

Theorem 9 (Extensivity)

If $(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)$ *then* $\exists \vec{y} d \vdash_{\mathcal{X}} \exists \vec{x} c$.

Proof.

For any constraint e, $c \wedge e \vdash_{\mathcal{X}} c$. □

Theorem 10 (Restartability)

If $(\vec{x}; c; \Gamma) \longrightarrow^* (\vec{y}; d; \Delta)$ *then* $(\vec{x}; \exists \vec{y} d; \Gamma) \longrightarrow^* (\vec{y}; d; \Delta)$.

Proof.

By extensivity and monotonicity. □

CC($\mathcal{X}$) Operational Semanticssss

- observing the set of success stores,
- observing the set of terminal stores (successes and suspensions),
- observing the set of accessible stores,
- observing the set of limit stores?

$$\mathcal{O}_\infty(\mathcal{D}.A; c_0) = \{\sqcup_? \{\exists \vec{x}_i c_i\}_{i \geq 0} | (\emptyset; c_0; A) \longrightarrow (\vec{x_1}; c_1; \Gamma_1) \longrightarrow \ldots\}$$

CC($\mathcal{X}$) Operational Semanticssss

- observing the set of success stores,

  $\mathcal{O}_{ss}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \; |(\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon)\}$

- observing the set of terminal stores (successes and suspensions),

- observing the set of accessible stores,

- observing the set of limit stores?

  $\mathcal{O}_\infty(\mathcal{D}.A; c_0) = \{\sqcup_? \{\exists \vec{x}_i c_i\}_{i \geq 0} | (\emptyset; c_0; A) \longrightarrow (\vec{x}_1; c_1; \Gamma_1) \longrightarrow \ldots\}$

CC($\mathcal{X}$) Operational Semanticssss

- observing the set of success stores,

$$\mathcal{O}_{ss}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon)\}$$

- observing the set of terminal stores (successes and suspensions),

$$\mathcal{O}_{ts}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma) \not\longrightarrow\}$$

- observing the set of accessible stores,

- observing the set of limit stores?

$$\mathcal{O}_{\infty}(\mathcal{D}.A; c_0) = \{\sqcup_? \{\exists \vec{x}_i c_i\}_{i \geq 0} | (\emptyset; c_0; A) \longrightarrow (\vec{x_1}; c_1; \Gamma_1) \longrightarrow \ldots\}$$

CC($\mathcal{X}$) Operational Semanticssss

- observing the set of success stores,

$$\mathcal{O}_{ss}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon)\}$$

- observing the set of terminal stores (successes and suspensions),

$$\mathcal{O}_{ts}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma) \not\longrightarrow\}$$

- observing the set of accessible stores,

$$\mathcal{O}_{as}(\mathcal{D}.A; c) = \{\exists \vec{x} d \in \mathcal{X} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; B)\}$$

- observing the set of limit stores?

$$\mathcal{O}_{\infty}(\mathcal{D}.A; c_0) = \{\sqcup_? \{\exists \vec{x}_i c_i\}_{i \geq 0} \mid (\emptyset; c_0; A) \longrightarrow (\vec{x}_1; c_1; \Gamma_1) \longrightarrow \ldots\}$$

INRIA

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

$$\begin{aligned} append(A, B, C) = {} & tell(A = [])||tell(C = B) \\ & + tell(A = [X|L])||tell(C = [X|R])||append(L, B, R) \end{aligned}$$

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

$$append(A, B, C) = tell(A = []) || tell(C = B)$$
$$+ tell(A = [X|L]) || tell(C = [X|R]) || append(L, B, R)$$

Directional CC success store style

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

$$append(A,B,C) = tell(A=[])||tell(C=B)$$
$$+tell(A=[X|L])||tell(C=[X|R])||append(L,B,R)$$

Directional CC success store style

$$append(A,B,C) = (A=[] \rightarrow tell(C=B))$$
$$+\forall X,L\ (A=[X|L] \rightarrow tell(C=[X|R])||append(L,B,R))$$

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

$$append(A, B, C) = tell(A = []) || tell(C = B) + tell(A = [X|L]) || tell(C = [X|R]) || append(L, B, R)$$

Directional CC success store style

$$append(A, B, C) = (A = [] \rightarrow tell(C = B)) + \forall X, L\ (A = [X|L] \rightarrow tell(C = [X|R]) || append(L, B, R))$$

Directional CC terminal store style

CC($\mathcal{H}$) 'append' Program(s)

Undirectional CLP style

$$append(A, B, C) = tell(A = []) || tell(C = B)$$
$$+ tell(A = [X|L]) || tell(C = [X|R]) || append(L, B, R)$$

Directional CC success store style

$$append(A, B, C) = (A = [] \rightarrow tell(C = B))$$
$$+ \forall X, L\ (A = [X|L] \rightarrow tell(C = [X|R]) || append(L, B, R))$$

Directional CC terminal store style

$$append(A, B, C) = A = [] \rightarrow tell(C = B)$$
$$|| \forall X, L\ (A = [X|L] \rightarrow tell(C = [X|R]) || append(L, B, R))$$

CC($\mathcal{H}$) 'merge' Program

Merging streams

$$
\begin{aligned}
merge(A,B,C) = {} & (A = [] \rightarrow tell(C = B)) \\
& +(B = [] \rightarrow tell(C = A)) \\
& +\forall X, L(A = [X|L] \rightarrow tell(C = [X|R]) || merge(L,B,R)) \\
& +\forall X, L(B = [X|L] \rightarrow tell(C = [X|R]) || merge(A,L,R))
\end{aligned}
$$

Good for the   observable(s?)

Many-to-one communication:
$client(C1, ...)$
...
$client(Cn, ...)$
$server([C1, ..., Cn], ...) =$
$$\sum_{i=1}^{n} \forall X, L(Ci = [X|L] \rightarrow ...||server([C1, ..., L, ..., Cn], ...)$$

CC($\mathcal{H}$) 'merge' Program

Merging streams

$$merge(A,B,C) = (A = [] \rightarrow tell(C = B))$$
$$+(B = [] \rightarrow tell(C = A))$$
$$+\forall X, L(A = [X|L] \rightarrow tell(C = [X|R])||merge(L,B,R))$$
$$+\forall X, L(B = [X|L] \rightarrow tell(C = [X|R])||merge(A,L,R))$$

Good for the $\mathcal{O}_{ss}$ observable

Many-to-one communication:
$client(C1, ...)$
...
$client(Cn, ...)$
$server([C1, ..., Cn], ...) =$
$$\sum_{i=1}^{n} \forall X, L(Ci = [X|L] \rightarrow ...||server([C1, ..., L, ..., Cn], ...)$$

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if $\mathcal{FD} \models c\sigma$ for every valuation σ of the variables in c by values of their domain.

Suppose access to *min* and *max* indexicals:
$ask(X \geq Y + k)$

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if $\mathcal{FD} \models c\sigma$ for every valuation σ of the variables in c by values of their domain.

Suppose access to *min* and *max* indexicals:

$$ask(X \geq Y + k) \qquad \cong min(X) \geq max(Y) + k$$

$$asknot(X \geq Y + k)$$

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if $\mathcal{FD} \models c\sigma$ for every valuation σ of the variables in c by values of their domain.

Suppose access to *min* and *max* indexicals:

$$ask(X \geq Y + k) \quad \cong min(X) \geq max(Y) + k$$

$$asknot(X \geq Y + k) \quad \cong max(X) < min(Y) + k$$

$$ask(X \neq Y)$$

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if $\mathcal{FD} \models c\sigma$ for every valuation σ of the variables in c by values of their domain.

Suppose access to *min* and *max* indexicals:

$$ask(X \geq Y + k) \cong min(X) \geq max(Y) + k$$

$$asknot(X \geq Y + k) \cong max(X) < min(Y) + k$$

$$ask(X \neq Y) \cong max(X) < min(Y) \vee min(X) > max(Y)$$

a better approximation with *dom*:

CC($\mathcal{FD}$) Finite Domain Constraints with indexicals

Approximating *ask* condition with the Elimination condition

EL: $c \wedge \Gamma \longrightarrow \Gamma$
if $\mathcal{FD} \models c\sigma$ for every valuation σ of the variables in c by values of their domain.

Suppose access to *min* and *max* indexicals:

$ask(X \geq Y + k) \quad \cong min(X) \geq max(Y) + k$

$asknot(X \geq Y + k) \quad \cong max(X) < min(Y) + k$

$ask(X \neq Y) \quad \cong max(X) < min(Y) \vee min(X) > max(Y)$
a better approximation with *dom*:
$\cong (dom(X) \cap dom(Y) = \emptyset)$

CC($\mathcal{FD}$) Constraints as "*in*.."

Basic constraints
$(X \geq Y + k) =$

CC($\mathcal{FD}$) Constraints as "*in..*"

Basic constraints

$(X \geq Y + k) = \quad X\ in\ min(Y) + k\ ..\ \infty \parallel Y\ in\ 0\ ..\ max(X) - k$

Reified constraints

$(B \Leftrightarrow X = A) =$

CC($\mathcal{FD}$) Constraints as "*in..*"

Basic constraints

$(X \geq Y + k) = \quad X\ in\ min(Y) + k\ ..\ \infty \ || \ Y\ in\ 0\ ..\ max(X) - k$

Reified constraints

$(B \Leftrightarrow X = A) = \quad B\ in\ 0..1\ ||$

CC($\mathcal{FD}$) Constraints as "*in..*"

Basic constraints

$(X \geq Y + k) = \quad X\ in\ min(Y) + k\ ..\ \infty\ ||\ Y\ in\ 0\ ..\ max(X) - k$

Reified constraints

$$\begin{aligned}(B \Leftrightarrow X = A) = \quad & B\ in\ 0..1\ ||\\ & X = A \rightarrow B = 1\ ||\ X \neq A \rightarrow B = 0\ ||\\ & B = 1 \rightarrow X = A\ ||\ B = 0 \rightarrow X \neq A\end{aligned}$$

Higher-order constraints

$card(N, L) =$

CC($\mathcal{FD}$) Constraints as "*in*.."

Basic constraints

$(X \geq Y + k) = \quad X\ in\ min(Y) + k\ ..\ \infty \ ||\ Y\ in\ 0\ ..\ max(X) - k$

Reified constraints

$$
\begin{aligned}
(B \Leftrightarrow X = A) = \quad & B\ in\ 0..1\ || \\
& X = A \rightarrow B = 1\ ||\ X \neq A \rightarrow B = 0\ || \\
& B = 1 \rightarrow X = A\ ||\ B = 0 \rightarrow X \neq A
\end{aligned}
$$

Higher-order constraints

$card(N, L) = \quad L = [] \rightarrow N = 0\ ||$

CC($\mathcal{FD}$) Constraints as "*in..*"

Basic constraints

$$(X \geq Y + k) = \quad X \; in \; min(Y) + k \; .. \; \infty \; || \; Y \; in \; 0 \; .. \; max(X) - k$$

Reified constraints

$$\begin{aligned}(B \Leftrightarrow X = A) = \quad & B \; in \; 0..1 \; || \\ & X = A \rightarrow B = 1 \; || \; X \neq A \rightarrow B = 0 \; || \\ & B = 1 \rightarrow X = A \; || \; B = 0 \rightarrow X \neq A\end{aligned}$$

Higher-order constraints

$$\begin{aligned}card(N, L) = \quad & L = [] \rightarrow N = 0 \; || \\ & L = [C|S] \rightarrow \\ & \exists B, M \; (B \Leftrightarrow C \; || \; N = B + M \; || \; card(M, S))\end{aligned}$$

Andora Principle

"Always execute deterministic computation first".

Disjunctive scheduling:

deterministic propagation of the disjunctive constraints for which one of the alternatives is dis-entailed:

$$card(1,\ [x \geq y + d_y,\ \ y \geq x + d_x])$$

before creating choice points:

$$(x \geq y + d_y) + (y \geq x + d_x)$$

Constructive Disjunction in CC($\mathcal{FD}$) (1)

$$\vee L \quad \frac{c \vdash_{\mathcal{X}} e \qquad d \vdash_{\mathcal{X}} e}{c \vee d \vdash_{\mathcal{X}} e}$$

Intuitionistic logic tells us we can *infer the common information* to both branches of a disjunction without creating choice points!

$max(X, Y, Z) = (X > Y || Z = X) + (X <= Y || Z = Y)$

or

$max(X, Y, Z) = X > Y \rightarrow Z = X + X <= Y \rightarrow Z = Y.$

or

$max(X, Y, Z) = X > Y \rightarrow Z = X \ || \ X <= Y \rightarrow Z = Y.$

better? (with indexicals)

Constructive Disjunction in CC($\mathcal{FD}$) (1)

$$\vee L \quad \frac{c \vdash_{\mathcal{X}} e \qquad d \vdash_{\mathcal{X}} e}{c \vee d \vdash_{\mathcal{X}} e}$$

Intuitionistic logic tells us we can *infer the common information* to both branches of a disjunction without creating choice points!

$max(X, Y, Z) = (X > Y || Z = X) + (X <= Y || Z = Y)$
or
$max(X, Y, Z) = X > Y \rightarrow Z = X + X <= Y \rightarrow Z = Y.$
or
$max(X, Y, Z) = X > Y \rightarrow Z = X \ || \ X <= Y \rightarrow Z = Y.$
better? (with indexicals)
$max(X, Y, Z) = Z \ in \ min(X)..\infty \ || \ Z \ in \ min(Y)..\infty$
$\quad || \ Z \ in \ dom(X) \cup dom(Y) \ || \ \cdots$

Constructive Disjunction in CC($\mathcal{FD}$) (2)

Disjunctive precedence constraints

$$
\begin{aligned}
\textit{disjunctive}(T1, D1, T2, D2) = \\
(T1 >= T2 + D2) + \\
(T2 >= T1 + D1)
\end{aligned}
$$

Using constructive disjunction

Constructive Disjunction in CC($\mathcal{FD}$) (2)

Disjunctive precedence constraints

$$disjunctive(T1, D1, T2, D2) = (T1 >= T2 + D2) + (T2 >= T1 + D1)$$

Using constructive disjunction

$$disjunctive(T1, D1, T2, D2) = T1\ in\ (0..max(T2) - D1) \cup (min(T2) + D2..\infty) \ || \ T2\ in\ (0..max(T1) - D2) \cup (min(T1) + D1..\infty)$$

Bibliography I

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

Vijay A. Saraswat.
Concurrent constraint programming.
ACM Doctoral Dissertation Awards. MIT Press, 1993.