A Domain-Independent Framework for the Analysis of the CCP Language AKL

-o-o-o-

presented at
Seminar on Programming Language Implementation
Stockholm
30 Sep 1997

Thomas Sjöland
SICS, Sweden

Analysis of AKL Programs

Domain-independent framework for a CCP language

- AKL and its computation model
- Computation rules and their abstract versions

The CCP Language AKL

[Haridi, Janson, Franzén]

Definition	::=	*Literal* `:=`	$\{V_1\} : G_1 \% B_1;$
			$\ldots$
			$\{V_n\} : G_n \% B_n.$
G_i	::=	*Literals*	
B_i	::=	*Literals*	
Literals	::=	*Literal* [,*Literals*]	
%	::=	`->` | `|` | `?`	
Literal	::=	*Name*(*Var*, ... *Var*) | *ConstrLit*	
ConstrLit	::=	*Var* `=` *Term* | constraint literal	
Term	::=	*Functor* [(*Var*, ..., *Var*)]	

Highlights of AKL

- implicitly concurrent computations
- computation controlled by constraint entailment
- logical variables, constraint literals (e.g. `X=f(Y)`)
- quietness in guards gives nested constraint stores
- allows deep guards, i.e. recursive formulation of guards
- non-determinate steps are postponed

E.g. (with the local variables explicit)

```
foo(A,B) :=
       {}: A=f(B)                  -> b1(B,A);
    {C,D}: g1(A,C), g2(B), c(A) -> b2(C,D), b3(A,C,D).
```

Guard Operators

`->` conditional
(ordered choice, requires entailment of guard)

`|` commit
(unordered choice, requires entailment of guard)

`?` wait
(generates alternatives for non-determinate choice,
"noisy", i.e. doesn't require entailment,
postponed until stable)

Quiet/Noisy Guards, Entailment

A computation step is enabled if

a noisy guard is solved and determinate
a quiet guard is solved, determinate
and entailed by its environment

otherwise when
a configuration is stable (non-determinate steps)

AKL is Concurrent, And/Or-Parallelism

```
def(X,Y,Z) :=
        {} : t(X,Y), r(Y,Z) -> p(X,Y), q(Y,Z).
```

Goals in guards/bodies may execute concurrently
also without suspension
although guards are always finished before entering the body

- Concurrency implicit through possibility of interleaving
- Analysis frameworks must allow interleaving
- No fixed ordering of goals in guards/bodies

Concrete and Abstract Semantics

Computation model [Janson94, Franzén94] (simplified here)

Rewrite of configurations form *computations*

$$\zeta_0 \longrightarrow \ldots \zeta_i \longrightarrow \ldots$$

An *abstract computation* formed by $\longrightarrow_\alpha$
is a bounded "sequence" of abstract configurations.

Both systems are indeterministic, i.e. we need to consider *trees* where each path to a leaf describes a possible computation.

The abstract paths may contain cycles.

Concrete and Abstract Computation Rules

$\chi[G]$ denotes a configuration χ containing G

$G \stackrel{\chi}{\Rightarrow} G'$ denotes the application of a rule

For sub-configurations G, G' we have

$\chi[G] \longrightarrow \chi[G']$ iff $G \stackrel{\chi}{\Rightarrow} G'$

and similarly for abstract computations

$\chi[G] \longrightarrow_\alpha \chi[G']$ iff $G \stackrel{\chi}{\Rightarrow}_\alpha G'$

Abstracting Constraints

Constraints, *Constr*, are sets of constraint literals,
e.g. θ=$\{x_1$=$f(y_2), y_3 = g(z_1, y_4)\}$

Abstract constraints, *AConstr*, are defined to cover constraints.
e.g. Θ=$\{x$=$f(y), y = g(z, y)\}$

Order relations $\preceq_{Constr}$ and $\preceq_{AConstr}$ form domains.

Functions:

$$\alpha : Constr \rightarrow AConstr$$

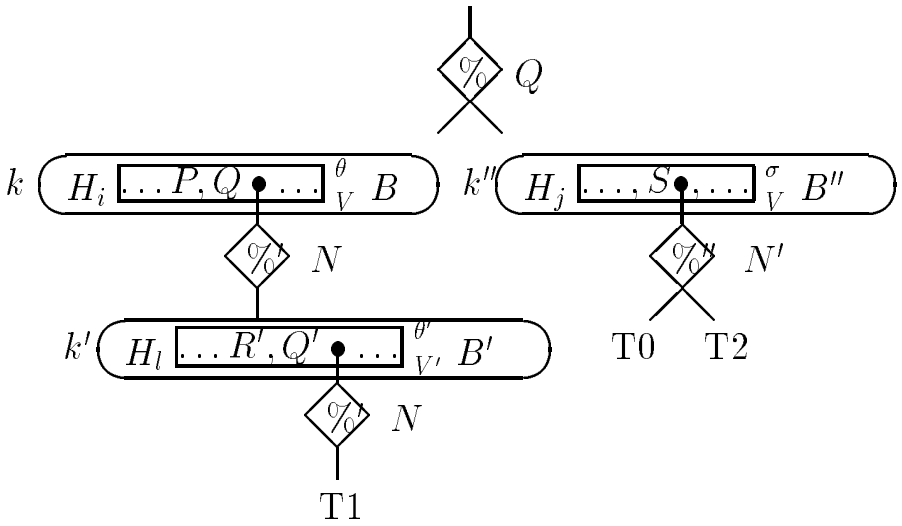
$$\gamma : AConstr \rightarrow Constr$$

Safeness:

$$\forall \theta \in Constr : \theta \preceq_{Constr} \gamma(\alpha(\theta))$$

$$\forall \Theta \in AConstr : \alpha(\gamma(\Theta)) =_{AConstr} \Theta$$

An Example of a Configuration



Configurations

$$
\begin{array}{rcl}
\langle \mathit{conf} \rangle & ::= & \langle \mathit{literal} \rangle \mid \langle \mathit{choice\text{-}box} \rangle \\
\langle \mathit{choice\text{-}box} \rangle & ::= & \mathbf{choice}(N, \%, \langle \text{sequence of } \langle \mathit{and\text{-}box} \rangle \rangle) \\
\langle \mathit{and\text{-}box} \rangle & ::= & \mathbf{and}(k, H_i, (\langle \text{sequence of } \langle \mathit{conf} \rangle \rangle)^{\theta}_{V} \langle \mathit{body} \rangle)
\end{array}
$$

in a choice-box:

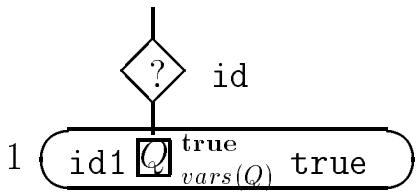
- N is a definition head literal
- $\%$ is a guard operator

In an and-box:

- k is a unique identifier
- H_i is a clause head literal
- V is a set of (local) variables
- θ is a (local) constraint

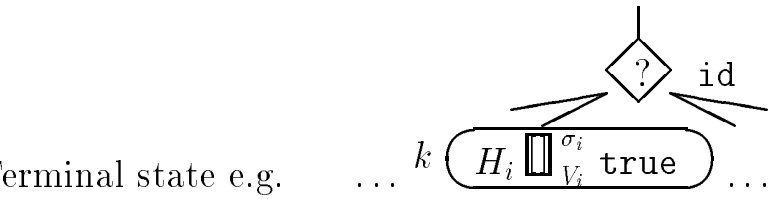
Initial Configuration

Computations start from:



where the literal Q is referred to as the *query*

Final Configurations

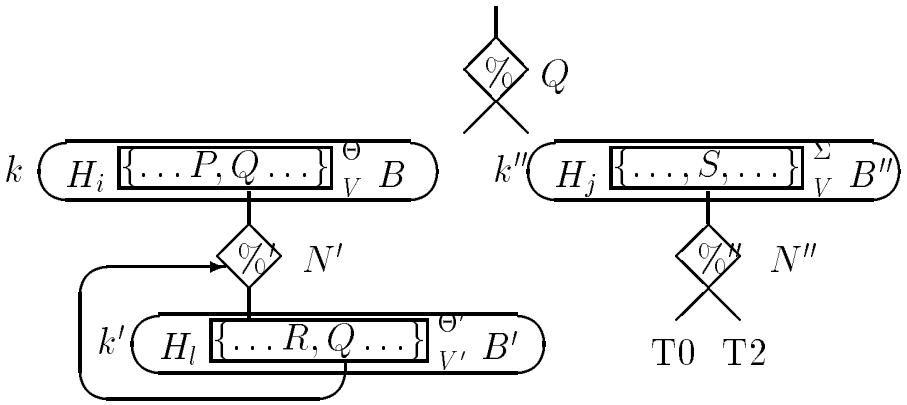
Terminal state e.g. 

represents at least one solution

The Computation Model Doesn't Specify:

- **choice of literal to resolve**
  no left-to-right sequencing of conjunctions,
  enables and-parallelism
- **choice-box to use for choice splitting**
  no left-to-right sequencing of search for alternatives,
  enables or-parallelism

Abstracting a Configuration



Cf. abstract and/or-trees in PLAI for Prolog
[Bruynooghe, Janssens, et.al.]

Abstract Configurations

- Sequences in concrete configurations are abstracted by sets
- The constraints in abstract **and**-boxes are abstracted by abstract constraints (bounded in size)
- Recursive calls (recurring choice-boxes of the form $x_i = c(\ldots x_j \ldots)$) are abstracted as $(x = c(\ldots x \ldots))$
- Concurrency is modeled by not removing literals

Abstracting Configurations

Configurations, *Conf*.

Abstract configurations, *AConf*, are defined to cover *Conf*.

Order relations $\preceq_{Conf}$ and $\preceq_{AConf}$ form domains.

Functions:

$$\alpha : Conf \to AConf$$

$$\gamma : AConf \to Conf$$

Safeness:

$$\forall C \in Conf : C \preceq_{Conf} \gamma(\alpha(C))$$

$$\forall A \in AConf : \alpha(\gamma(A)) =_{AConf} A$$

Safe abstract computation model

An *abstract computation* formed by $\longrightarrow_\alpha$
is a bounded "sequence" of abstract configurations

$$\alpha(\zeta_0) \longrightarrow_\alpha \ldots \alpha(\zeta_i) \longrightarrow_\alpha \ldots$$

For all $\zeta_i \longrightarrow \zeta_{i+1}$: $\alpha(\zeta_i) \longrightarrow_\alpha A$

$$\text{we have } \alpha(\zeta_{i+1}) \preceq_{Abs} A$$

and for all $A \longrightarrow_\alpha A'$: $\forall \zeta \in \gamma(A), (\zeta \longrightarrow \zeta')$)

$$\text{we have } \zeta' \preceq_{Conc} \gamma(A')$$

i.e. the abstract rules *cover* the concrete ones.

(Tedious but transparent proof by cases and induction.)

Generic Domain Functions

- **bot**, the lowest domain element
- **or**(S) computes the least upper bound of the domain
- **conj**(S) models a concurrent computation
- **constraint**(G) models the abstraction of a predefined literal G
- **proj**(Θ, H, C) models the propagation of constraints from an abstract constraint Θ wrt the literals C and H
- **restrict**(V, Θ) models the set of constraints modeled by Θ limited to the variables V

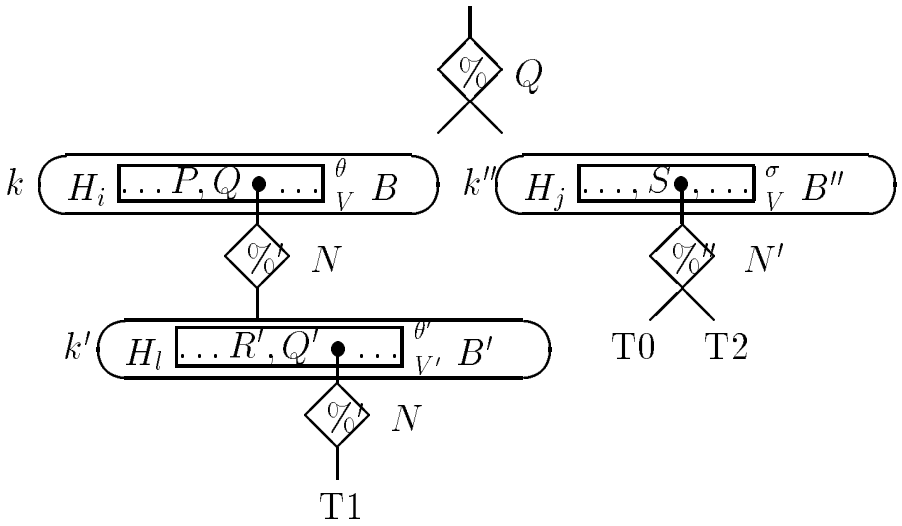
Environments

conjunctions of constraints "visible" from
a part of a configuration
$store(k, \chi) = \theta$ where
$\chi[\mathbf{and}(k, _, (_)^{\theta}_{V}, _)]$

$father(k, k', \chi)$ where
$\chi[\mathbf{and}(k', _, (\ldots \mathbf{choice}(_, _, \ldots \mathbf{and}(k, _, _, _)\ldots)\ldots)\underline{\ }, _)]$

- $env(k, \chi) = store(k, \chi)$ if $\neg\exists k' : father(k, k', \chi)$
- $env(k, \chi) = store(k, \chi) \wedge env(k', \chi)$ where $father(k, k', \chi)$ otherwise

Environment in a configuration

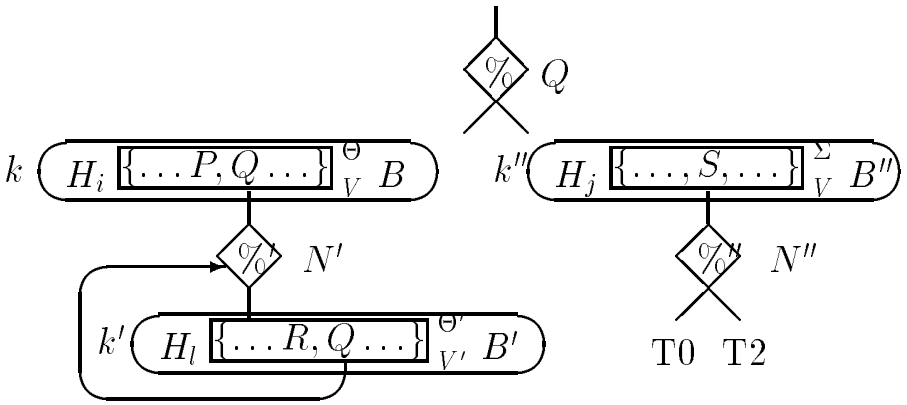


$store(k', \chi) = \theta'$
$father(k', k, \chi)$
$env(k', \chi) = \theta' \wedge \theta$

Abstract Environments

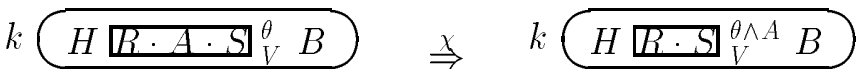
- $env_\alpha(i,\chi) = store(i,\chi)$ if $\neg\exists i' : father(i,i',\chi)$
- $env_\alpha(i,\chi) = \mathbf{conj}(store(i,\chi) \cup \{env_\alpha(i',\chi) \mid father(i,i',\chi)\})$ otherwise

Abstract environment in a configuration



$store(k', \chi) = \Theta'$
$father(k', k, \chi)$
$father(k', k', \chi)$
$env_\alpha(k', \chi) = \mathbf{conj}(\{\Theta', \Theta\})$

Constraint Literal Rule

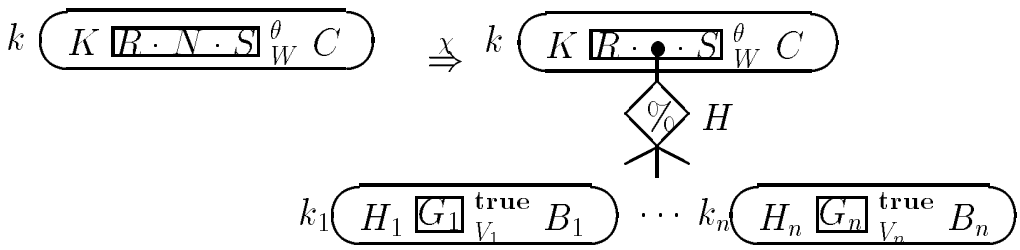


adds a constraint literal A to the local constraint store

Abstract Constraint Literal Rule

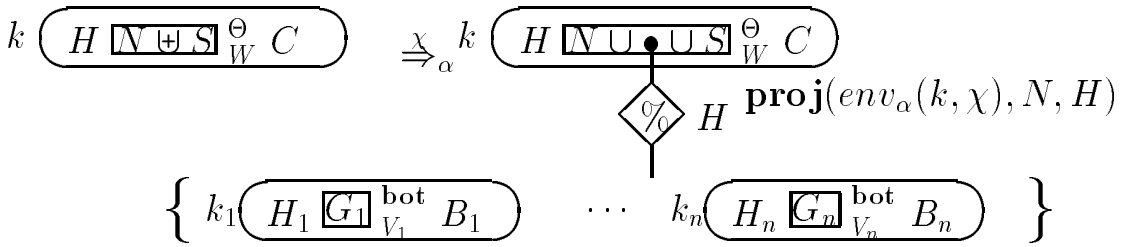
The domain function expression $\mathbf{conj}(\Sigma, \mathbf{constraint}(A))$ and not removing the constraint literal A takes care of the modeling of permutations of the literals in the surrounding and-box (Either A is executed or not)

Unfolding Rule



N is rewritten using a new instance of the clauses defining H, $H_i : -V_i' : G_i \ \% \ B_1 \cdot \ldots \cdot H_n : -V_n' : G_n \ \% \ B_n$.
The arguments of N are substituted for the head parameters in each H_i.
The local variables, V_i', of the i:th clause are replaced by the variables in V_i.
The sets V_i are chosen to be disjoint from each other and from the variables of G.

Abstract Unfolding Rule

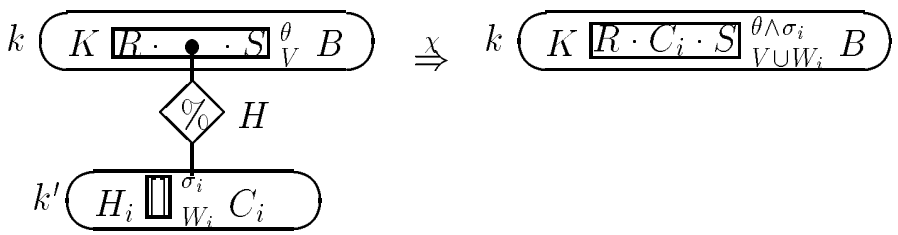


The arguments of N are substituted for the head parameters in each H_i. V_i are the local variables of the clause (i) used.
The rule is only applied if an essentially new abstract environment is produced. Otherwise the abstract configuration represents a rational tree (circular graph).

Precondition for the definition of H:

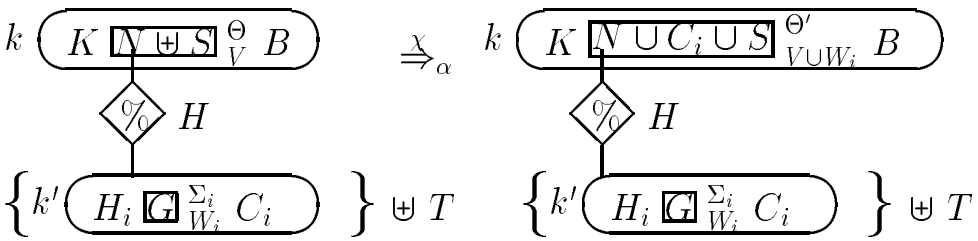
$$\mathbf{proj}(env_\alpha(k,\chi), N, H)$$

Promotion Rule



promotes a guarded goal with a single solved guard and-box. If % is '$\rightarrow$' or '|' it is required that σ_i is quiet with respect to $env(k, \chi)$ and V.

Abstract Promotion Rule



promotes a guarded abstract goal with a solved guard abstract and-box.

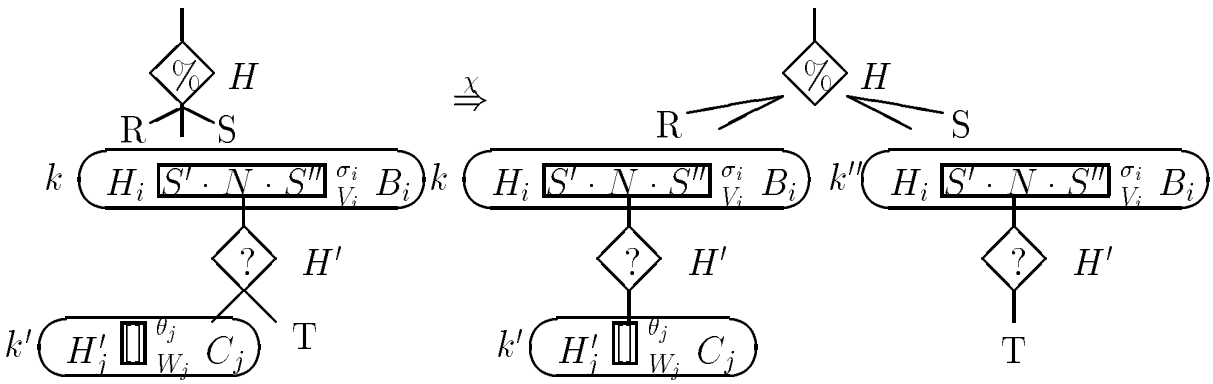
The abstract environment of the and-box:
if (% =?) then $\Theta' = \mathbf{conj}(\Theta, \mathbf{proj}(\mathbf{proj}(\Sigma_i, H_i, H), H, N))$
else $\Theta' = \mathbf{conj}(\Theta, \mathbf{proj}(\mathbf{proj}(\mathbf{restrict}(W_i, \Sigma_i), H_i, H), H, N))$
here covering also the quiet case.

Other Rules

The rule system of AKL contains some additional rules that are of no concern here since they do not contribute to the set of abstract environments. They can be modeled by a rule

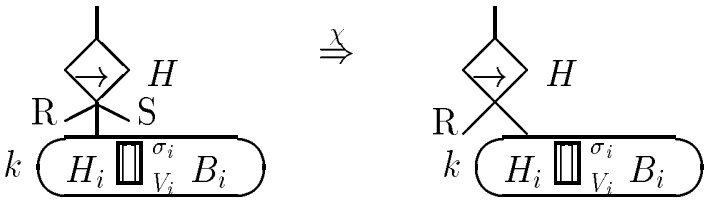
$$G \overset{\chi}{\Rightarrow}_{\alpha} G$$

Choice Splitting Rule



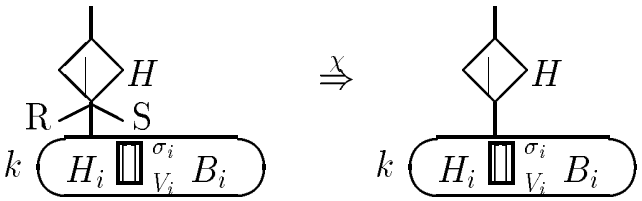
distributes nondeterminism in an inner nondeterminate choice-box over an and-box. T must be non-empty and the rewritten and-box must be a sub-configuration of a *stable* configuration.

Conditional Pruning Rule



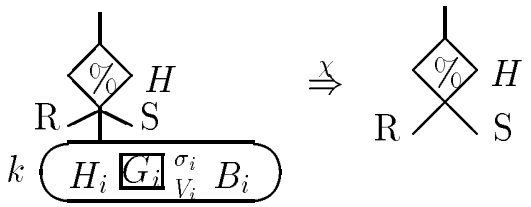
may be applied if S is non-empty and σ_i is quiet with respect to $env(k', \chi)$ and V_i for $father(k, k', \chi)$.

Commit Pruning Rule



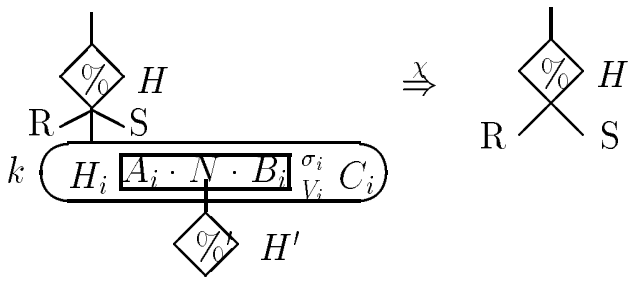
may be applied if at least one of R and S is non-empty and σ_i is quiet with respect to $env(k', \chi)$ and V_i for $father(k, k', \chi)$.

Environment Failure Rule



fails an and-box if σ_i and $env(k', \chi)$ for $father(k, k', \chi)$ are incompatible.

Goal Failure Rule



removes an and-box if it contains an empty choice-box.

Conclusions

The computation of the analysis determines a covering abstract configuration
This is achieved through a fixpoint computation

The analysis

- is *polymorphic*
- treats *concurrency*
- treats *quiet/noisy guards*
- is *independent of the constraint system*

(for a simpler model see ParForce, [Sahlin,Sjöland,Brand,Schön])

Possible Improvements

- Higher-order constructs, feature structures, ports
- Identifying sequential threads
- Modeling definite groundness after a guard
- Modeling pruning
- Separate analysis of parts of programs
- "real" languages (CLP, Prolog, Oz, Erlang, Java,...)