A Domain-Independent Framework for the Analysis of the CCP Language AKL
(extended abstract)

Per Brand[1] and Thomas Sjöland[1]

Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden

Abstract. A domain-independent framework for the analysis of the concurrent constraint programming language AKL is presented. It is based on an abstraction of the operational semantics of AKL. Issues of synchronisation and pruning are not treated.

1 Introduction

Analysis works for concurrent logic programming languages have occurred e.g. in [11, 24, 12, 14, 10, 28], all of which relating to flat languages. In the ESPRIT project ParForce we developed a framework for analysis of type and alias in AKL programs [21]. The framework presented here is a generalisation of the one we presented in [4].

An analysis system for the concurrent constraint language AKL [16, 17] should model also concurrency and deep guards and not depend on the sequential execution order. All analysis results should be valid also for a parallel implementation of AKL [19] and for other constraint systems than term constraints [18, 7].

2 Definitions

Sequences are appended with "$\cdot$", e.g. $e_1 \cdot e_2 \cdot e_3$ is a sequence of three subsequences. [] (sometimes omitted) stands for an empty sequence. An element is also considered as a sequence.

Vars is a denumerably infinite set of variables $\{v_i | i \geq 0\}$. *Cons* is a denumerably infinite set of term constructors $\{c_i | i \geq 0\}$. Each term constructor has an *arity* ≥ 0. *Structs* is the set of *compound terms*, $N(t_1 \dots t_n)$, where $n = arity(N)$, $t_1 \dots t_n \in Vars$ and $N \in Cons$ and $arity(N(\bar{t})) = arity(N)$. $t \in Structs$ is called a *constant* if $arity(t) = 0$. $Terms = Vars \cup Structs$. *Clits* is the set of *constraint literals* containing at least the literals $\{v = t \mid v \in Vars \wedge t \in Terms\} \cup \{\mathbf{true}, \mathbf{false}\}$. A *constraint* σ is a set of constraint literals. A constraint θ is considered equivalent to $\theta \cup \{v = v\}$ for any variable v. A constraint where all literals (if any) are of the form $v = v$ can be written $\{\}$, $\{\mathbf{true}\}$ or simply **true**. A singleton set of constraint literals $\{A\}$ is also written A without risk of confusion.

$\exists\sigma$ stands for the existential closure of σ and $\exists V$ for (any permutation of) the quantifier sequence $\exists v_1 \ldots v_n$, where V is a set $\{v_1, \ldots, v_n\}$ of variables. We allow V to be empty, in which case $\exists V$ is mere decoration.

A constraint θ (understood as a conjunction) can be *entailed*, i.e. implied, by another constraint, σ, written $\sigma \supset \theta$.

Assume given some complete and consistent *constraint theory* **TC** the following defined logical properties of constraints:

1. σ is *satisfiable* iff $\mathbf{TC} \models \exists\sigma$.
2. σ is *quiet w.r.t. θ and V* iff $\mathbf{TC} \models \theta \supset \exists V\sigma$, where the variables in V do not occur in θ. A constraint which is not quiet is called *noisy*.
3. σ and θ are *incompatible* iff $\mathbf{TC} \models \neg(\sigma \wedge \theta)$.

$\mathbf{TC} \models$ **true** and $\mathbf{TC} \not\models$ **false**.

The expressions $\theta \wedge \sigma$ and $\theta \wedge A$ denote an extension of the constraint θ with another constraint σ or a constraint literal A. We use standard notions of *composition* (of constraints), i.e. $\sigma \wedge \theta \equiv \sigma\theta \equiv \sigma \cup \theta$, and *renaming* (wrt. two sequences of variables). *Substitution* is understood in terms of an *environment*, i.e. a constraint.

2.1 The Programming Language AKL

The concurrent constraint logic programming language [22] AKL, was first defined in [16]. A thorough description of the language and the rationale behind it is given in [17]. There is a treatment of various formal aspects of the language semantics in [15].

The abstract syntactic categories for program definitions are:

$\langle program \rangle ::= \langle query \rangle \langle$sequence of $\langle definition \rangle\rangle$
$\langle query \rangle ::= \langle literal \rangle$
$\langle definition \rangle ::= \langle dhead \rangle \langle clauses \rangle$
$\langle dhead \rangle ::= \langle program\ literal \rangle$
$\langle clauses \rangle ::= \langle$sequence of $\langle clause \rangle$ with the same guard operator$\rangle$
$\langle clause \rangle ::= \langle chead \rangle \langle$set of $Pvars\rangle \langle guard \rangle \langle guard\ operator \rangle \langle body \rangle$
$\langle chead \rangle ::= \langle program\ literal \rangle$
$\langle guard \rangle ::= \langle statement \rangle$
$\langle body \rangle ::= \langle statement \rangle$
$\langle statement \rangle ::= \langle literal \rangle \mid \langle literal \rangle \langle statement \rangle$
$\langle guard\ operator \rangle ::= '\rightarrow' \mid '\,|\,' \mid '?'$

Programs are normalized, i.e. all literals in clauses have corresponding definitions, variables occurring in a clause are disjunct from those in other clauses and from the query, structures occur only in constraint literals, variables in a literal are all disjunct and variables occurring in definition head literals are disjunct from those in all other literals.

The set $Pvars \subset Vars$ is the set of program variables, the set $Pstructs \subset Structs$ is the set of structures, and the set $Pterms \subset Terms$ is the set of terms occurring in the program. v or $Name(v_1 \ldots v_n)$ occurs in the program iff $v = Name(v_1 \ldots v_n)$ is a constraint literal in the program.

Each definition in a program is uniquely identified by a literal $H(\overline{x})$ ($\langle dhead \rangle$ in the grammar, omitted in the concrete syntax) and contains a sequence of clauses with the same guard operator. The guard operator determines the kind of definition as one of the following forms: conditional (`'->'`), commit (`'|'`) or wait (`'?'`). Variables in programs begin with capital letters, `Foo`.

The clauses representing the alternatives in a definition are each uniquely identified by a literal $H_i(\overline{x})$, ($\langle chead \rangle$). The $\langle$set of $Pvars\rangle$ of $\langle clause \rangle$ are the local variables in the clause, i.e. those variables that do not occur in the head literal.

A single literal $Q(\overline{x})$, ($\langle query \rangle$), representing the call of a definition that starts a computation. The analyses we consider must assume some entry point into the code so this leads to no loss of generality.

An important difference in concurrent constraint logic programming languages compared to Prolog is that subsets of deterministic literals can be executed in parallel. This allows a process view of programs in contrast to the procedural view in Prolog. Compared to other concurrent logic programming languages like KL1 [8] or FCP [26, 25], guard literals may be user-defined. Such guards are named deep guards and are also found in the languages GHC [27] and Parlog [9]. Another difference is the way in which AKL allows the order of clauses (of the conditional and wait- types) in a definition to express control. Conditional clauses are chosen based on the order of the clauses. In commit definitions the clause is chosen symmetrically, i.e. without reference to the order of clauses. The wait operator provides don't-know nondeterministic computation, while the conditionals and the commit respectively provide deterministic and don't-care non-deterministic computations.

3 Abstracting the AKL Semantics

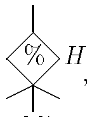
The semantic rules governing the AKL execution are formulated as a set of rewrite rules on goals, (containing and-, and choice-boxes). % stands for any guard operator.

3.1 Goals and Computations

We have the additional abstract syntactic categories for goals:

$$\langle goal \rangle ::= \langle literal \rangle \mid \langle choice\text{-}box \rangle$$
$$\langle choice\text{-}box \rangle ::= \mathbf{choice}(\langle dhead \rangle \;\%\; \langle \text{sequence of } \langle and - box \rangle \rangle)$$
$$\langle and\text{-}box \rangle ::= \mathbf{and}(\langle ident \rangle \langle chead \rangle (\langle \text{sequence of } \langle goal \rangle \rangle)^{\langle constraint \rangle}_{\langle \text{set of } Vars \rangle} \langle body \rangle)$$

Goals are constraint literals, user-defined literals or choice-boxes,

$\mathbf{choice}(H, \%\,, \langle \text{sequence of } \langle and - box \rangle \rangle)$, graphically , with a guard operator, %. Choice-boxes also contain a literal, H, the head literal of the definition used to generate the choice-box, and a sequence of and-boxes, $\mathbf{and}(k, H_i, (G_i)^{\theta_i}_{V_i}, B_i)$, graphically $^{k}\left(H_i\ \boxed{G_i}{}^{\theta_i}_{V_i}\ B_i \right)$, or are possibly empty, $\mathbf{choice}(H, \%, [])$, graphically $\diamond\!\%\ H$. The G_i in an and-box is a sequence of goals. And-boxes also contain a literal, H_i, showing the head literal of the clause (i) used to generate the and-box, a unique and-box identifier k, and another sequence of literals, B_i, representing the body of the clause used to construct the and-box (in the unfolding rule, see below). The and-box is adorned with a local constraint θ_i and a set, V_i, of local variables.

Either a literal, a choice-box or an and-box with their respective contents can be considered a *subgoal* to the current goal χ. $\chi[G]$ denotes a goal expression χ containing G as a subgoal.
$store(k, \chi) = \theta$ where $\chi[\mathbf{and}(k, \_, (\_)^{\theta}_{V}, \_)]$.
$father(k, k', \chi)$ where $\chi[\mathbf{and}(k', \_, (\ldots \mathbf{choice}(\_, \_, \ldots \mathbf{and}(k, \_, \_, \_) \ldots) \ldots)^{\_}_{\_}, \_)]$.

Since each and-box in a goal χ (except the topmost box) has a unique *father*, the and-box immediately above it, the environment of a goal can be defined through its unique identifier k:
$env(k, \chi) = store(k, \chi)$ if $\neg\exists k' : father(k, k', \chi)$
$env(k, \chi) = store(k, \chi) \ \wedge \ env(k', \chi)$ where $father(k, k', \chi)$ otherwise.

The Concrete and Abstract Computation Models
Computations are possibly non-deterministic and possibly infinite sequences of goals, $\chi_0 \rightarrow \chi_1 \rightarrow \ldots \rightarrow \chi_m \rightarrow \ldots$ formed with the computation rules introduced below.

An abstract model $\rightarrow_\alpha$ is defined similarly. Abstracted versions of literals are achieved by using the program variables that gave rise to the variables occurring in the literal. The and-boxes are adorned with abstract constraints. In the abstract boxes the sequences of goals correspond to sets.

Relation to the Concrete Semantics
The abstract computation system $\rightarrow_\alpha$ abstracts the concrete one $\rightarrow$ in that any step $\zeta \rightarrow \zeta'$ is abstracted by some $\alpha(\zeta) \rightarrow_\alpha \alpha(\zeta')$. The *safeness* of the abstract computation model follows since the abstract model is conservative wrt. the concrete computation model, i.e. in other words any goal that can occur in an execution using the concrete computation model is a member of the concretisation of some abstract goal generated by the abstract computation model. The *termination* of the abstract computations is guaranteed since both (1) the abstract goals and (2) the abstract computations are bounded in size, (since only finite sets of known constraint literals are possible, each sequence of abstract unfoldings which doesn't introduce new variables will sooner or later

lead to a repeated occurrence of the same abstract environment, thus allowing a bounded representation of the possibly unbounded goal/computation). The formal verification of the safeness and termination are omitted in the current paper.

3.2 The Domain Functions

We consider only a number of generic domain functions, which are used in the abstract semantic rules. The intention of the domain functions is the following:

- **bot** is the lowest domain element.
- **or**(S) computes the least upper bound of the domain for the elements in the set S. This function models results from alternative branches of the computation.
- **concs**(S) is a composition of the values of elements in S that models any partial, i.e. concurrent, computation of a group of literals.
- **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 set of variables V.

3.3 Abstract Environments, Initial and Final Goals

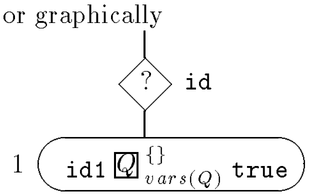
The *abstract environment* of an abstract goal with an identifier i, is defined thus:
$env_\alpha(i, \chi) = store(i, \chi)$ if $\neg\exists i' : father(i, i', \chi)$
$env_\alpha(i, \chi) = \mathbf{concs}(store(i, \chi) \ \cup \{env_\alpha(i', \chi) \ \mid \ father(i, i', \chi)\})$ otherwise

Since an abstract goal may be a rational tree (contain cycles), the computation of env_α is done as a fixpoint computation.

The abstract computation rules form abstract computations, where sequences in goals are unordered sets, and all variables are program variables. The abstract rule set can be evaluated with a fixpoint iteration. The choice-boxes of the program's definitions found in the final abstract goal are also adorned with abstract constraints.

Initial Goal

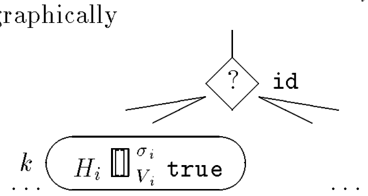
The initial goal is a choice box with a single alternative of the form
$\mathbf{choice}(\mathtt{id}, ?, \mathbf{and}(1, \mathtt{id1}, (Q)^{\{\}}_{vars(Q)}, \mathtt{true}))$
or graphically



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

Final Goals

When no more rule is applicable the current goal represent the terminal state e.g. $\mathbf{choice}(\mathtt{id}, ?, \ldots \mathbf{and}(k, H_i, ()^{\sigma_i}_{V_i}, \mathtt{true}) \ldots)$
or graphically



represents at least one solution.

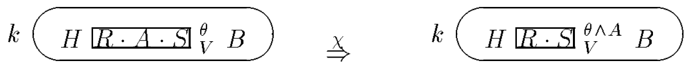
3.4 The Concrete and Abstract Rules

We show the computation rules and in the relevant cases also their abstract counterparts. For simplicity we present the abstract rules annotated with the current domain functions, although a slightly more general form is required for the rules to be completely domain independent.

For subgoals G, G' we have $\chi[G] \to \chi[G']$ iff $G \overset{\chi}{\Rightarrow} G'$ and similarly for abstract computations $\chi[G] \to_\alpha \chi[G']$ iff $G \overset{\chi}{\Rightarrow}_\alpha G'$.

Constraint Literal Rule

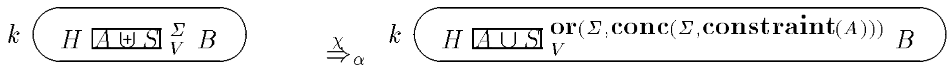
$\mathbf{and}(k, H, (R \cdot A \cdot S)^{\theta}_{V}, B) \overset{\chi}{\Rightarrow} \mathbf{and}(k, H, (R \cdot S)^{A \wedge \theta}_{V}, B)$
or graphically



moves a constraint literal A to the constraint part of the and-box, thereby making it part of the constraint of the goals in the box.

Abstract Constraint Literal Rule

$\mathbf{and}(k, H, (A \uplus S)^{\Sigma}_{V}, B) \overset{\chi}{\Rightarrow}_\alpha \mathbf{and}(k, H, (A \cup S)^{\mathbf{or}(\Sigma, \mathbf{conc}(\Sigma, \mathbf{constraint}(A)))}_{V}, B)$
or graphically

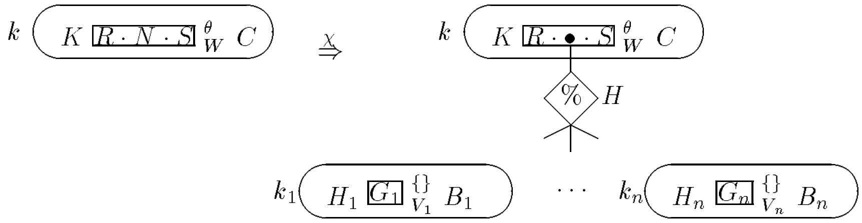


The domain function expression $\mathbf{or}(\Sigma, \mathbf{conc}(\Sigma, \mathbf{constraint}(A)))$ and not removing the constraint literal A takes care of the modeling of permutations of the literals in the surrounding and-box. (Either A is executed or not.)

Unfolding Rule

$$\mathbf{and}(k, K, (R \cdot N \cdot S)^{\theta}_{W}, C) \overset{\chi}{\Rightarrow}$$
$$\mathbf{and}(k, K, (R \cdot \mathbf{choice}(H, \%, \mathbf{and}(k_1, H_1, (G_1)^{\{\}}_{V_1}, B_1) \cdot \ldots \cdot \mathbf{and}(k_n, H_n, (G_n)^{\{\}}_{V_n}, B_n)) \cdot S)^{\theta}_{W}, C)$$

or graphically

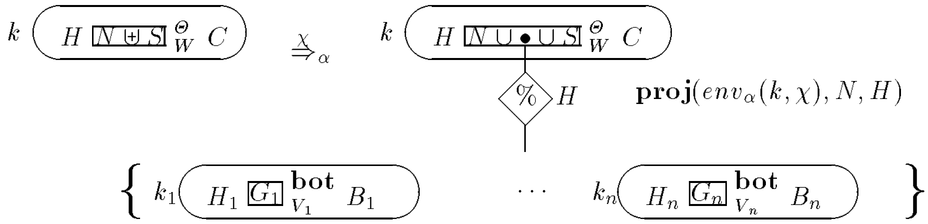


where a program literal subgoal N is rewritten using a new instance of the sequence of clauses defining H, $H_i : -V_i' : G_i \ \% \ B_1 \cdot \ldots \cdot H_n : -V_n' : G_n \ \% \ B_n$. The arguments of N are substituted (wrt. the current environment) for the head parameters in each H_i. The local variables, V_i', of the i:th clause are replaced by the variables in the set V_i. When a goal G is rewritten to G' by applying unfolding, the sets V_i are chosen to be disjoint from each other and from the set of variables of G.

Abstract Unfolding Rule

$$\mathbf{and}(k, K, (N \uplus S)^{\Theta}_{W}, C) \overset{\chi}{\Rightarrow}_{\alpha}$$
$$\mathbf{and}(k, K, (\{N, \mathbf{choice}(H, \%, \{\mathbf{and}(k_1, H_1, (G_1)^{\mathbf{bot}}_{V_1}, B_1) \cdot \ldots \cdot \mathbf{and}(k_n, H_n, (G_n)^{\mathbf{bot}}_{V_n}, B_n)\}) \} \cup S)^{\Theta}_{W}, C)$$

or graphically



The arguments of N are substituted (wrt. the current abstract environment) for the head parameters in each H_i, and the local variables of the i:th clause are replaced by the variables in the set V_i. When a goal G is rewritten to G' by applying unfolding, the sets V_i are given by the local variables in the program clause used. Not removing the literal N takes care of the modeling of permutations of the literals in the surrounding and-box. (Either N is executed or not.)

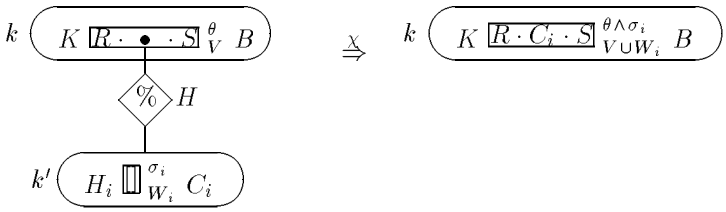
The rule is only applied if an essentially new abstract environment is produced. Otherwise the *father*-relation is updated for the unfolding candidate N to make the abstract goal represent a rational tree (circular graph).

We get the equation corresponding to the definition of H:
$\mathsf{Pre\_}H = \mathbf{proj}(env_{\alpha}(k,\chi), N, H)$

Promotion Rule

$\mathbf{and}(k, K, (R \cdot \mathbf{choice}(H, \%\ , \mathbf{and}(k', H_i, ()^{\sigma_i}_{W_i}, C_i)) \cdot S)^{\theta}_{V}, B) \stackrel{\chi}{\Rightarrow}$
$\mathbf{and}(k, K, (R \cdot C_i \cdot S)^{\theta \wedge \sigma_i}_{W_i \cup V}, B)$

or graphically

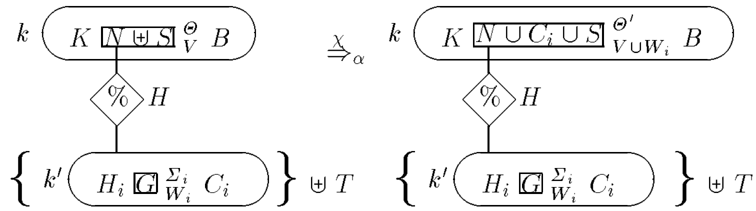


promotes a guarded goal with a single solved guard and-box. If $\%$ is '$\rightarrow$' or '$|$' it is required that σ_i is quiet with respect to $env(k,\chi)$ and V.

Abstract Promotion Rule

$\mathbf{and}(k, K, (\{\mathbf{choice}(H, \%, \{\mathbf{and}(k', H_i, (G)^{\Sigma_i}_{W_i}, C_i)\} \uplus T)\} \uplus S)^{\Theta}_{V}, B) \stackrel{\chi}{\Rightarrow}_{\alpha}$
$\mathbf{and}(k, K, (\{\mathbf{choice}(H, \%, \{\mathbf{and}(k', H_i, (G)^{\Sigma_i}_{W_i}, C_i)\} \uplus T)\} \cup C_i \cup S)^{\Theta'}_{V \cup W_i}, B)$

or graphically



promotes a guarded abstract goal with a solved guard abstract and-box.
We get the equation determining the abstract environment of the and-box:
if ($\% =?$) then $\Theta' = \mathbf{or}(\Theta, \mathbf{conc}(\Theta, \mathbf{proj}(\mathbf{proj}(\Sigma_i, H_i, H), H, N)))$
else $\Theta' = \mathbf{or}(\Theta, \mathbf{conc}(\Theta, \mathbf{proj}(\mathbf{proj}(\mathbf{restrict}(W_i, \Sigma_i), H_i, H), H, N)))$
here covering also the quiet case.

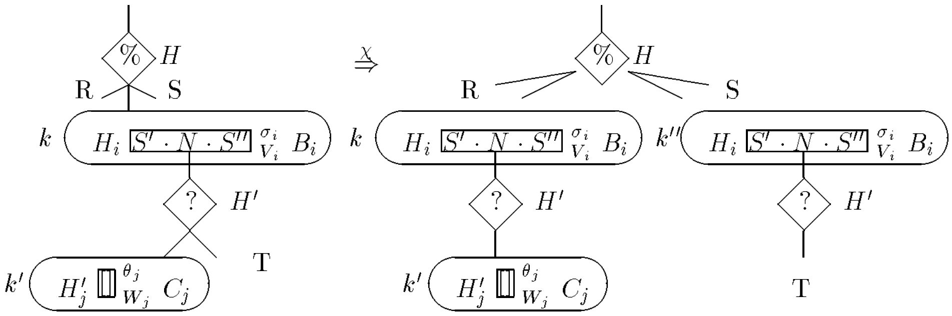
3.5 Other Rules

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 $G \overset{\chi}{\Rightarrow}_{\alpha} G$. We list the concrete versions for completeness.

Choice Splitting Rule

$$\begin{array}{l}
\mathbf{choice}(H, \%\ , R \cdot \\
\qquad \mathbf{and}(k, H_i, (S' \cdot \mathbf{choice}(H'\ , ?, \mathbf{and}(k', H'_j, ()^{\theta_j}_{W_j}, C_j) \cdot T) \cdot S'')^{\sigma_i}_{V_i}, B_i) \\
\qquad \cdot S) \\
\qquad\qquad \overset{\chi}{\Rightarrow} \\
\mathbf{choice}(H, \%\ , R \cdot \\
\qquad \mathbf{and}(k, H_i, (S' \cdot \mathbf{choice}(H', ?, \mathbf{and}(k', H'_j, ()^{\theta_j}_{W_j}, C_j)) \cdot S'')^{\sigma_i}_{V_i}, B_i), \\
\qquad \mathbf{and}(k'', H_i, (S' \cdot \mathbf{choice}(H', ?, T) \cdot S'')^{\sigma_i}_{V_i}, B_i) \\
\qquad \cdot S)
\end{array}$$

or graphically



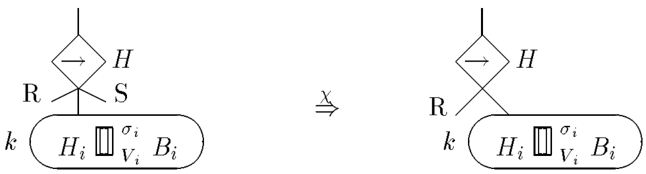
distributes nondeterminism in an inner nondeterminate choice-box over an and-box, creating alternatives in an outer box. T must be non-empty and the rewritten and-box must be a subgoal of a *stable* goal.[1]

Choice splitting is said to be performed *with respect to* the and-box $\mathbf{and}(k', H_j, ()^{\theta_j}_{W_j}, C_j)$, and in a goal matching the left hand side of the rule. The and-box is called a *candidate* for choice splitting.

Pruning Rules

$$\mathbf{choice}(H, \rightarrow, R \cdot \mathbf{and}(k, H_i, ()^{\sigma_i}_{V_i}, B_i) \cdot S) \overset{\chi}{\Rightarrow} \mathbf{choice}(H, \rightarrow, R \cdot \mathbf{and}(k, H_i, ()^{\sigma_i}_{V_i}, B_i))$$

or graphically

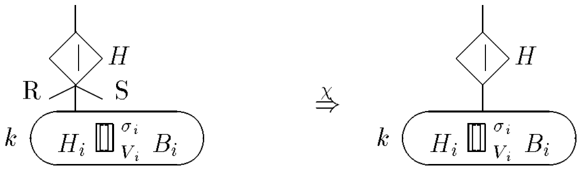


may be applied if S is non-empty and σ_i is quiet with respect to $env(k', \chi)$ and V_i for $father(k, k', \chi)$.

---

[1] The notion of stability, a condition used to postpone the application of the choice splitting rule, is explained in both [17, 15] and is not treated here.

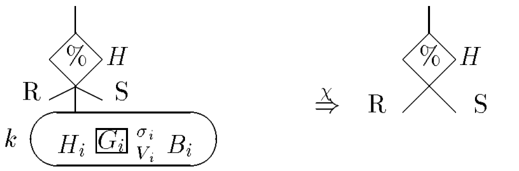
$\mathbf{choice}(H, |\ , R \cdot \mathbf{and}(k, H_i, ()^{\sigma_i}_{V_i}, B_i) \cdot S) \overset{\chi}{\Rightarrow} \mathbf{choice}(H, |\ , \mathbf{and}(k, H_i, ()^{\sigma_i}_{V_i}, B_i))$
or graphically



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 $father(k, k', \chi)$.

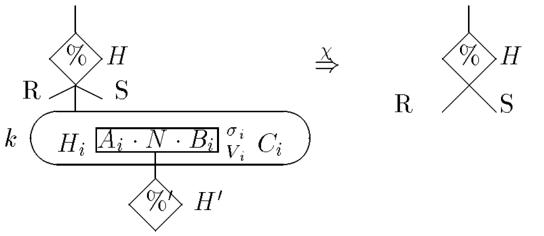
Failure Rules

$\mathbf{choice}(H, \%\ , R \cdot \mathbf{and}(k, H_i, (G_i)^{\sigma_i}_{V_i}, B_i) \cdot S) \overset{\chi}{\Rightarrow} \mathbf{choice}(H, \%\ , R \cdot S)$
or graphically



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

$\mathbf{choice}(H, \%\ , R \cdot \mathbf{and}(k, H_i, (A_i \cdot \mathbf{choice}(H'\ \%') \cdot B_i)^{\sigma_i}_{V_i}, C_i) \cdot S) \overset{\chi}{\Rightarrow}$
$\mathbf{choice}(H, \%\ , R \cdot S)$
or graphically



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

3.6 Conclusion

The framework is *polymorphic* and treats quietness. Experiments are required to show whether this approach gives a practical analyser in terms of efficiency.

Acknowledgments

We acknowledge good advice from Dan Sahlin and Per Kreuger.

References

1. R. Barbuti, R. Giacobazzi and G. Levi. A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. *ACM Trans. on Programming Languages and Systems*, 15(1):133-181, 1993.
2. Per Brand and Seif Haridi. Andorra Prolog: The Language and Application in Distributed Simulation. Technical report, SICS, 1988. IC5GCS, Tokyo.
3. P. Brand, D. Sahlin and T. Sjöland. An Analyser for a Concurrent Constraint Language. (poster). Abstract in *Proc. of. the International Conference of Logic Programming.* Kanagawa, Japan, June 1995. MIT Press, Ed. L. Sterling.
4. P. Brand, D. Sahlin, E. Schön and T. Sjöland. Efficient Analysis of Type and Alias Information for a Concurrent Constraint Language. In *Analysis and Verification of Multiple-Agent Languages. Fifth LOMAPS Workshop. Abstracts.* Eds. Mads Dam and Fredrik Orava. SICS Research Report R96:05, ISRN SICS-R–96/05–SE.
5. P. Brand. Optimization for AKL with Global Analysis. Deliverable D.WP.2.1.3.M2 ParForce, Ed. M. Hermenegildo, 1994.
6. P. Brand. A Decision Graph Algorithm for CCP Languages. In *Proc. of. the International Conference of Logic Programming.* Kanagawa, Japan, June 1995.
7. B. Carlsson. Compiling and Executing Finite Domain Constraints. Uppsala Theses in Computer Science 21, ISSN 0283-359X, ISBN 91-506-1100-3, Uppsala University, and SICS Dissertation Series 18, ISRN SICS/D-18–SE, ISSN 1101-1335.
8. T. Chikayama. Parallel Inference System in the Fifth Generation Computer System Institute for New Generation Computer Technology, Japan.
9. K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.
10. M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In *Proc. of the 20th International Colloquium on Automata, Languages, and Programming*, volume 700 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.
11. Christian Codognet, Philippe Codognet, and Marc-Michel Corsini. Abstract Interpretation for Concurrent Logic Languages. In Hermenegildo and Debray, editors, *North American Conference on Logic Programming*, pages 215–232, 1990.
12. Vítor Santos Costa. Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I. Ph.D Thesis. Dept. of Computer Science, Bristol University, UK, 1993.
13. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. in *Journal of Logic Programming*, 13(2&3):103-180, 1992.
14. Saumya K. Debray, David Gudeman and Peter Bigot. Detection and Optimization of Suspension-free Logic Programs. In *Proceedings of International Logic Programming Symposium*, 1994. pages 487–501, MIT Press.
15. T. Franzén. Some Formal Aspects of AKL. SICS Research Report R94:10, ISRN SICS R–94/10–SE.
16. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. in *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991.
17. S. Janson. AKL - A Multiparadigm Programming Language. Uppsala Theses in Computer Science 19, ISSN 0283-359X, ISBN 91-506-1046-5, Uppsala University, and SICS Dissertation Series 14, ISRN SICS/D-14–SE, ISSN 1101-1335.

18. T. Keisu. Tree Constraints. The Royal Institute of Technology (KTH), TRITA-IT 94:10, ISSN 1103-534X, and SICS Dissertation Series 16, ISRN SICS/D-16–SE, ISSN 1101-1335.
19. J. Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. In *New Generation Computing*, 14 (1996) 31-52, Ohmsa, LTD and Springer Verlag.
20. K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):291-314, 1992.
21. D. Sahlin and T. Sjöland. Fixpoint Analysis of Type and Alias in AKL Programs. SICS Research Report R94:13, ISRN SICS-R–94/13–SE.
22. V. A. Saraswat. Concurrent Constraint Programming Languages. Ph.D. Thesis 1989, published by MIT Press 1993.
23. E. Schön. On the Computation of Fixpoints in Static Program Analysis with an Application to Analysis of AKL. Masters Thesis, KTH, Stockholm, Sweden, 1995.
24. R. Sundarajan, A. V. S. Sastry and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *JICLP92*, pp493–508.
25. Akikazu Takeuchi and Koichi Furukawa. Parallel Logic Programming Languages. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, pages 242–254, London, July 1986.
26. S. Taylor, S. Safra, and E. Shapiro. A Parallel Implementation of Flat Concurrent Prolog. Internal CS87-04, Weizmann Institute of Science, October 1986.
27. Kazunori Ueda. *Guarded Horn Clauses.* Ph.D. Thesis, University of Tokyo, March 1986.
28. E. Zaffanella, G. Levi and R. Giacobazzi. Abstracting Synchronization in Concurrent Constraint Programming. Univ. di Pisa, Italy and LIX, Paris, France.