-o-o-o-

Efficient Analysis of Type and Alias for a Concurrent Constraint Language

-o-o-o-

presented at Lomaps'96

Per Brand, Dan Sahlin*, Erik Schön**, *Thomas Sjöland*

SICS
* Computer Science Laboratory, Ericsson Telecom
** Ericsson Radio Systems

Analysis of Parallel AKL programs

Part of the ESPRIT Project ParForce (finished jan 96)

Goal:
To provide the new AKL compiler [Brand] with information to make the programs run faster and consume less memory.

Overview

- AKL and its computation model
- framework
- analysed properties
- domain
- examples
  - type and alias at definition pre-points
  - reusable data structures
- efficiency
- conclusions

The CCP language AKL

[Haridi, Janson, Franzén]

$$
\begin{array}{lll}
Definition & ::= & Literal \; \texttt{:=} \; Vars_i : G_1 \% B_1; \\
& & \quad \ldots \\
& & \quad Vars_n : G_n \% B_n. \\
Vars_i & ::= & \text{set of } Var \\
G_i & ::= & Literals \\
B_i & ::= & Literals \\
Literals & ::= & Literal \; [,Literals] \\
\% & ::= & \texttt{->} \mid \texttt{|} \mid \texttt{?} \\
Literal & ::= & Name(Var, \ldots Var) \mid CLit \\
CLit & ::= & Var \; \texttt{=} \; Term \\
Term & ::= & Functor \; [\; (Var, \ldots, Var) \;]
\end{array}
$$

E.g. (with the local variables explicit)

```
append(A,B,C) :=
       {}: A=[]    -> B=C;
  {H,T,R}: A=[H|T] -> append(T,B,R), C=[H|R].
```

Highlights of AKL

- logical variables
- implicitly concurrent subcomputations
- deep guards (encapsulated subcomputations in guards)
- non-determinate steps are postponed

Guard operators

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

`|` commit
(unordered choice, requires entailment)

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

Quiet/noisy guards, entailment

A computation step is enabled if
a quiet guard is entailed by its environment

a configuration is stable

Operational semantics

Computation model [Janson94, Franzén94] (simplified here)

Rewrite of configurations form computations

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

where ζ_i are configurations and $\longrightarrow$ is defined through rewrite rules on configurations.

Configurations are trees of **choice**- and **and**-boxes, **and**-boxes are decorated with constraints

Syntactic notion of constraints

Constraint literals $v = t$ are elementary operations where

v is a variable

and

t is a variable or a term $f(v_1 ... v_n)$

A constraint is a (normalized) set of constraint literals

E.g.:

$$\{X = f(Y), Y = g(Z), Z = [H|T]\}$$

The constraint $\{\}$ is named **true**.

Normalisation rules

u, v are variables, s, t are any terms, θ is a constraint.

- 1: $s = t \in eq(\theta)$ if $s = t \in \theta$
- 2: $t = s \in eq(\theta)$ if $s = t \in eq(\theta)$
- 3: $s = t \in eq(\theta)$ if $v = s, v = t \in eq(\theta)$
- 4: $u = s \in eq(\theta)$ if $u = v, v = s \in eq(\theta)$
- 5: $s_1 = t_1, \ldots, s_n = t_n \in eq(\theta)$
  if $f(s_1 \ldots s_n) = f(t_1 \ldots t_n) \in eq(\theta)$
- 6: $v = v \in eq(\theta)$

Entailment

Entailment $\theta \models \sigma$ used in the computation model considers a constraint as a conjunctive formula of equalities.

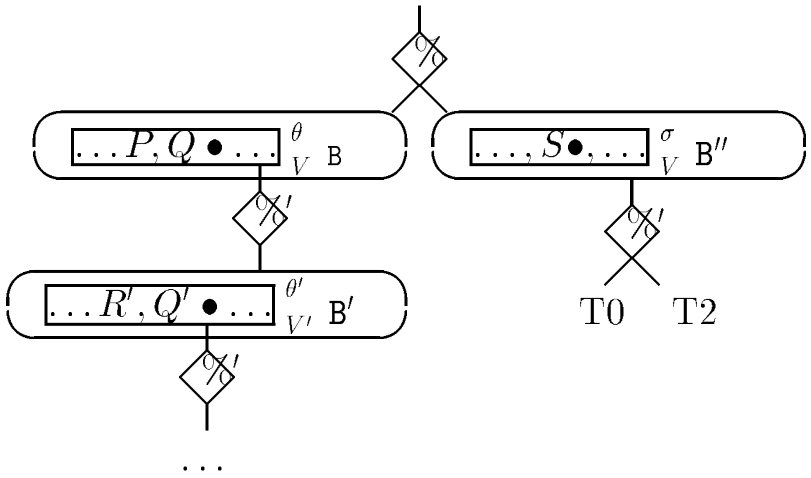
$$\{X = f(Z), Y = f(Z)\} \models \{X = Y\}$$

while

$$\{X = Y\} \not\models \{X = f(Z), Y = f(Z)\}$$

The analysis collects descriptions of all constraint literals that can occur the environments of a computation (and some more).

A configuration

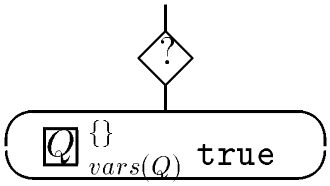


Collecting semantics

All possible computations starting from Q, an initial query:

$$\mathbf{choice}(?, \mathbf{and}((Q)^{\{\}}_{vars(Q)}, \texttt{true}))$$

or graphically



AKL is concurrent, and/or-parallelism

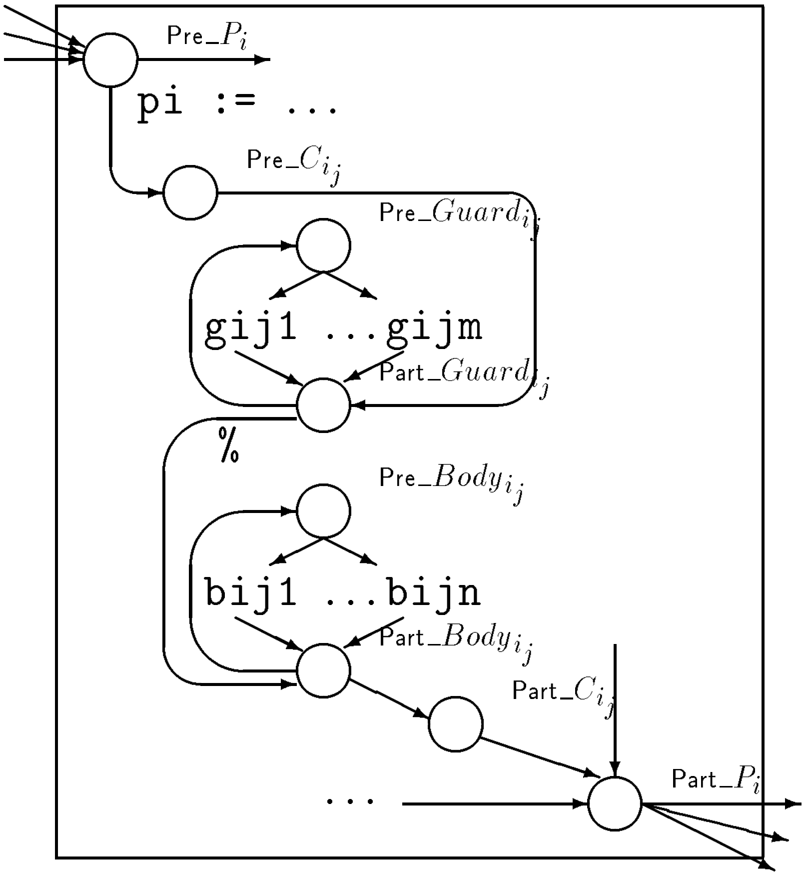
```
def(X,Y,Z) :=
        {} : t(X,Y), r(Y,Z) -> p(X,Y), q(Y,Z).
```

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

Analysis frameworks must allow interleaving.

Loop-back

The framework uses a loop-back propagating the results from the exit of each literal to the input of the **and**-box containing the literal.



The dataflow of an AKL definition.

Motivation of the framework

Firstly

- relax the rules to ignore stability, pruning and failure.
- treat all guard operators as the noisy wait (`?`)

Then given a computation

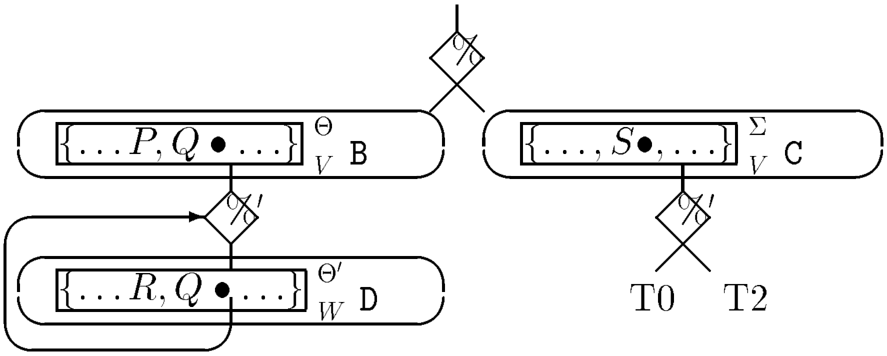
$$\zeta_0 \longrightarrow_s \ldots \zeta_i \longrightarrow_s \ldots$$

where ζ_i are configurations and $\longrightarrow_s$ is defined through the relaxed rewrite rules on configurations.

An abstract computation formed by $\longrightarrow_\alpha$ is a bounded sequence of abstract configurations.
For all $\zeta_i \longrightarrow_s \zeta_{i+1}$

$$\alpha(\zeta_i) \longrightarrow_\alpha \alpha(\zeta_{i+1})$$

Abstracting a configuration



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

Abstract configurations

- Sequences occurring in concrete configurations are abstracted by sets.
- The constraints adorning the abstract **and**-boxes are abstracted by domain elements (bounded in size).
- Recursive call literals (recurring choice-boxes) are modeled by memoizing tabulation.

A table of programs points keeps domain equations modeling the connections between parts of the environment.

E.g.
The output of a choice-box is fed back into the input of its surrounding **and**-box (loop-back to model concurrency).

Properties of interest

- $alias(X, Y, \theta)$

  there is an X-variable v_1 and a Y-variable v_2 such that the constraint literal $v_1 = v_2$ occurs in the environment θ.

- $in\_type(X, f(Y, \dots W), \theta)$

  there is an X variable v and Y-,...W-variables $v_1, \dots v_n$, such that $v = f(v_1, \dots v_n)$ occurs in the environment θ.

Domain for abstracting constraints

$$
\begin{array}{lcl}
D & ::= & \text{set of } Pair \\
Pair & ::= & \langle Alias \ - \ Type \rangle \\
Alias & ::= & \text{set of } Variable \\
Type & ::= & \text{set of } Value \\
Value & ::= & Name(Arg_1, \ldots Arg_n) \ \mid \ \textbf{number} \ \mid \ \ldots \\
Arg_i & ::= & \text{set of } Variable
\end{array}
$$

- A variable occurs in exactly one $Alias$ set
- Each $Type$ contains at most one occurrence of each structure with a particular name, $Name/n$

On aliases

- an *alias set* is a set of abstract variables, $(\{X, Y\})$
- a *type* is a set of descriptions of possible values, $(\{f(\{Z\})\})$

In an abstraction of the constraint

$$\{X_1 = f(Z_3), Y_2 = f(Z_3)\}$$

$$\{\langle\{X\} - \{f(\{Z\})\}\rangle, \langle\{Y\} - \{f(\{Z\})\}\rangle\}$$

X and Y are NOT aliased,
while the logically (but not syntactically) equivalent constraint

$$\{X_1 = f(Z_3), Y_2 = X_1\}$$

is represented by

$$\{\langle\{X, Y\} - \{f(\{Z\})\}\rangle\}$$

Concretisation

E.g.

$$
\begin{aligned}
&\gamma(\{\langle\{X,Y\} - \{f(\{Z,Y\})\}\rangle \\
&\ \langle\{Z\} - \{g(\{X\})\}\rangle \\
&\ \langle\{U\} - \{a\}\rangle\})
\end{aligned}
$$

$$=$$

$$
\begin{aligned}
\wp(\{&X_1 = X_3, Y_2 = X_2, \\
&Z_2 = X_3, U_4 = U_7, X_5 = X_6, \\
&X_5 = f(Z_8), Y_9 = f(Z_1), \\
&X_8 = f(U_5), Y_2 = f(U_3), \\
&Z_2 = g(X_7), U_3 = a \\
&\dots \\
\}&)
\end{aligned}
$$

A description representing a constraint also represents all weaker constraints.

Use of the analysis result

The compiler can use:

- X and Y are not aliased
  (avoids testing at run-time)
- type specification for values of a variable
  (allows simplified decision code for clause selection)
- identification of reusable structures

Examples from the deliverable

```
fig4.qsort(
        X1 ::= [F|X1] | [] | _X1
        X2 ::= [F|X2] | [] | _X2
        ). with
        F  ::= _F | number
```

[Cf. vanHentenryck et.al JLP 1995 p186]

An example: insert/3

```
insert(E,S,T) :=
        S=void
    ->  L=void, R=void, T=tree(L,E,R);

        S=tree(L,V,R), E=<V
    ->  T=tree(Ln,V,R),  insert(E,L,Ln);

        S=tree(L,V,R), E>V
    ->  T=tree(L,V,Rn), insert(E,R,Rn).
```

```
ex1b.insert(
        E ::=   number | _E
        S ::=   tree(S,E,S) | void | _S
        T ::=   tree(S,E,S) | void | _S | _T
        ).
```

Analysis of reusable structures

For variables with more than 2 occurrences in a clause the value sets are extended with a marker **'m'** after analysis of type and alias.

'm' is propagated downwards into all descriptions of possible subterms.

$$
\begin{aligned}
&\{\langle\{X,Y\} - \{f(\{Z\},\{V\})\}\rangle\\
&\ \langle\{Z\} - \{g(\{U\}),\mathtt{m}\}\rangle\\
&\ \langle\{U\} - \{\}\rangle\\
&\ \langle\{V\} - \{t(\{U\})\}\rangle\}
\end{aligned}
$$

gives

$$
\begin{aligned}
&\{\langle\{X,Y\} - \{f(\{Z\},\{V\})\}\rangle\\
&\ \langle\{Z\} - \{g(\{U\}),\mathtt{m}\}\rangle\\
&\ \langle\{U\} - \{\mathtt{m}\}\rangle\\
&\ \langle\{V\} - \{t(\{U\})\}\rangle\}
\end{aligned}
$$

Instances of the variables Z and U are possibly unsafe, others are safe.

Example of derived unsafe references

```
ex1b.insert(
        E ::=    _E 'm'
        S ::=   tree(S,E,S) | void | _S
        T ::=   tree(S,E,S) | void | _S | _T
        ).
```

Domain Equations

From the set of abstract computations a set of domain equations defining the abstract environments for the program points can be extracted. (we actually generate these directly)

$$x_i = f_i(\overline{x})$$

which are solved with a general fix point solver.

Improving the solver

The first solver (SCC) was improved by the use of a result by Bourdoncle [E. Schön SICS-R95:06].

A weak topological ordering of the nodes can be performed giving an ordering of applications in a subproblem.

This gave appr. 3 times improvement in speed for most examples.
(Porting from AKL to SICStus Prolog gave appr. 5 more)

Performance Table

[performance table from Schön]

Conclusions

The analysis

- treats concurrency
- is adequately fast
  [cf vanHentenryck et.al JLP]
- is reasonably precise for its purpose
  [Brand, ParForce]

Possible improvements

- separate analysis of parts of programs and combination of these (for practical use with larger programs)
- modeling of nested environments (deep guards) i.e. distinction quiet/noisy
- modeling of definite groundness after a guard
- treatment of higher-order constructs, feature structures, other constraint systems than rational trees
- modeling of pruning