Tutorial on analysis and verification of imperative programs through CLP

John P. Gallagher\textsuperscript{1,2}

\textsuperscript{1}Roskilde University \quad \textsuperscript{2}IMDEA Software Institute, Madrid

ICLP 2015, Cork
Ireland

Acknowledgements
EU FP7 ENTRA Project
EU Coordination Action ICT-Energy
Danish Natural Science Council NUSA Project
Program verification – one of the earliest applications of predicate logic in computing.

Developed by some of the pioneers of computing including Turing, Hoare, Dijkstra, Floyd, Gries, ... .

A hard, "grand challenge" problem.
Logic programming frameworks for imperative program manipulation

Much work since the 1980s within the (C)LP community addressed various aspects of modelling, interpreting and verification of imperative programs.

(Too many to list. This is a tutorial, not a survey.)

NB. Verifying logic programs is as hard a problem as verifying any other programs.
Recently, constrained Horn clauses (CHCs) have been adopted by researchers (not in mainstream CLP) working in automatic software verification (software model checking). E.g. see


Other software verification researchers include Flanagan, Rybalchenko, Bjørner, Rümmer, Kuncak, Gurfinkel, Podelski, Kroening, Jackson, ... 

CHCs are syntactically and semantically identical to CLP programs, though CHCs are not necessarily viewed as executable programs.
Analysis and verification of imperative programs through CLP
Verification problems

- proving invariants, safety assertions,
- assume-guarantee proofs
- termination
- resource consumption

Only time to talk about the first two topics.
assume $P$ and assert $P$ can be placed anywhere in the code. Verification consists of checking that the assertions hold whenever they are encountered during execution, under the provided assumptions. Total correctness also requires verifying that the assertion are reached (termination of the program).

Example from Heizmann, Hoenicke, and Podelski 2013.
Often translation is treated in an ad hoc, "intuitive" way. But it is important

1. to be sure that the CLP clauses correctly express the verification problem

2. to obtain CLP clauses that are amenable to efficient analysis.

We briefly look at

- Semantics-based translation
- Intermediate code/Control-flow-graph translation
- Generation of verification conditions
Semantics-based translation from imperative programs to CLP

An imperative program $P$ defines a relation

$$s \langle P \rangle s'$$

where $s, s'$ are the states of the program variables before and after execution of $P$.

$s \langle P \rangle s'$ specified using operational semantics transitions.

Two main styles: big-step or small-step.
Big-step vs. Small-step

Given program statement $s$ and state $\sigma$,  

- **Big-step.** Transition is written $\langle s, \sigma \rangle \Rightarrow \sigma'$ which holds if $s$ executes completely in initial state $\sigma$ and terminates in final state $\sigma'$.

- **Small-step.** Transition $\langle s, \sigma \rangle \Rightarrow \langle s', \sigma' \rangle$ which holds if $s$ executes one step in state $\sigma$ and moves to the next statement $s'$ and next state $\sigma'$. 
Big-step transitions follow the syntactic structure of the program. E.g.

\[
\begin{align*}
\langle S_1, \sigma \rangle & \Rightarrow \sigma' \quad \langle S_2, \sigma' \rangle \Rightarrow \sigma'' \\
\langle S_1 ; S_2, \sigma \rangle & \Rightarrow \sigma''
\end{align*}
\]

\[
\begin{align*}
\langle s, \sigma \rangle & \Rightarrow \sigma' \quad \langle \text{while} \ (b) \ s, \sigma' \rangle \Rightarrow \sigma'' \\
\langle \text{while} \ (b) \ s, \sigma \rangle & \Rightarrow \sigma'' \quad \text{if } b \text{ is true in } \sigma
\end{align*}
\]

\[
\langle \text{while} \ (b) \ s, \sigma \rangle \Rightarrow \sigma \quad \text{if } b \text{ is false in } \sigma
\]
Small-step transitions are also syntax-driven but can construct new statements as continuations. E.g.

\[
\langle s_1, \sigma \rangle \Rightarrow \langle s'_1, \sigma' \rangle \\
\langle s_1 ; s_2, \sigma \rangle \Rightarrow \langle s'_1 ; s_2, \sigma' \rangle
given \ s'_1 \neq \text{skip}
\]

\[
\langle s_1, \sigma \rangle \Rightarrow \langle \text{skip}, \sigma' \rangle
given \ s'_1 \neq \text{skip}
\]

\[
\langle s_1 ; s_2, \sigma \rangle \Rightarrow \langle s_2, \sigma' \rangle
given \ s'_1 \neq \text{skip}
\]

where \( b \) is true in \( \sigma \)

\[
\langle \text{while} \ (b) \ s, \sigma \rangle \Rightarrow \langle s ; \text{while} \ (b) \ s, \sigma \rangle
given \ s'_1 \neq \text{skip}
\]

where \( b \) is false in \( \sigma \)

For small-step semantics, we also need a multi-step transition \( \Rightarrow^* \) that chains together the small steps.
Define a transition predicate corresponding to the semantic transitions.

- **Big-step**: \(\langle s, \sigma \rangle \Rightarrow \sigma'\) represented as \(\text{trans}(S, \text{Sigma}, \Sigma_1)\).

- **Small-step**: \(\langle s, \sigma \rangle \Rightarrow \langle s', \sigma' \rangle\) represented as \(\text{trans}(S, \text{Sigma}, S_1, \Sigma_1)\).
Semantic transitions as CLP clauses

Transitions translate straightforwardly to CLP clauses.

- Judgements (transitions)
  
  \[
  \frac{\alpha_1, \ldots, \alpha_n}{\alpha} \quad \text{where } b
  \]

- CLP
  
  \[
  \alpha :\!-\! \alpha_1, \ldots, \alpha_n, b.
  \]

Note that the definitions of side-conditions \( b \) can be "programmed" in CLP.
Small-step semantics: defining a run

- Multi-step computation

\[ \langle \text{skip}, \sigma \rangle \Rightarrow^* \langle \text{skip}, \sigma \rangle \]

\[ \langle s_0, \sigma_0 \rangle \Rightarrow \langle s_1, \sigma_1 \rangle \quad \langle s_1, \sigma_1 \rangle \Rightarrow^* \langle s_2, \sigma_2 \rangle \]

\[ \langle s_0, \sigma_0 \rangle \Rightarrow^* \langle s_2, \sigma_2 \rangle \]

- CLP

run(skip,St,skip,St).
run(S0,St0,S2,St2) :-
    trans(S0,St0,S1,St1),
    run(S1,St1,S2,St2),
Partial evaluation

Given a program $S$, partially evaluate (specialise) the semantics wrt. $S$.

- Write your own specialiser or use off-the-shelf tool: e.g. LOGEN (M. Leuschel)
- The CLP semantics interpreter is annotated to indicate
  - which calls are unfolded
  - a "filter" for each argument controlling generalisation and removal of static structures

- In small-step semantics, partially evaluate $\text{run}(S, \_\_, \_\_, \_\_\_)$; every call to $\text{trans}$ is unfolded, leaving the calls to $\text{run}$.
- In big-step semantics, partially evaluate $\text{trans}(S, \_\_, \_\_)$; calls to $\text{trans}$ are not unfolded (just flattened, see below).
Structure filtering

Standard "flattening" transformation – removes redundant structure and retains only the variables (dynamic values in LOGEN).

```/* run(stm(let(n,cns(nat(C))),let(m,cns(nat(D))),seq(ifnz(var(n),
    seq(seq(asg(m,mul(var(n),var(m))),asg(n,sub(var(n),cns(nat(E))))),
    while(var(n),seq(asg(m,mul(var(n),var(m))),
    asg(n,sub(var(n),cns(nat(F))))),skip),ret(var(m)))))),[],A,[],B) :-
run__3(A,F,C,D,E,B). */

run__3(A,F,C,D,E,B) :-
    run__4(A,F,C,D,C,E,G), B is 1+G.

Flattening can be formally justified using unfold-fold transformations.
```
Big-step vs. small-step semantics

The form of the semantics determines the form of the CLP program.

- Big-step semantics generally makes it easier to obtain a "good" partial evaluation since it follows the syntactic structure directly.
- Small-step semantics produces programs that are "transition systems" and are "easier to analyse"; but . . .
- For recursive programs, small-step semantics requires a stack to be represented explicitly in the CLP program.
- For big-step semantics, the stack is implicit in the CLP semantics.
- Big-step semantics needs to record the input and output state variables for each program part; small-step does not.
Consider the following nested loop.

```plaintext
i = 0;
while (i < n) do
    j = 0;
    while (j < m) do
        p(i,j); // procedure that might update i,j,n,m
        j = j+1;
    od
    i = i+1;
od
```
trans(N,M, N1,M1) :-
    I = 0,
    while1(I,N,M, I1,N1,M1).
while1(I,N,M, I,N,M) :-
    I >= N.
while1(I,N,M, I3,N2,M2) :-
    I < N,
    J = 0,
    while2(I,J,N,M, I1,J1,N1,M1),
    I2 = I1+1,
    while1(I2,N1,M1, I3,N2,M2).
while2(I,J,N,M, I,J,N,M) :-
    J >= M.
while2(I,J,N,M, I2,J3,N2,M2) :-
    J < M,
    p(I,J,N,M, I1,J1,N1,M1),
    J2 = J1+1,
    while2(I1,J2,N1,M1, I1,J3,N2,M2).
p(I,J,N,M, I1,J1,N1,M1) :- ...

The loops while1 and while2 are each recursive predicates. while1 calls while2
but not vice versa.
Small-step - example translation to CLP

run(N,M, N1,M1) :-
    I = 0;
    while1(I,N,M, N1,M1).
while1(I,N,M, N,M) :-
    I > N.
while1(I,N,M, N1,M1) :-
    I =< N,
    J = 0,
    while2(I,J,N,M, N1,M1).
while2(I,J,N,M, N2,M2) :-
    J > M,
    I1 = I+1,
    while1(I1,N,M, N1,M1).
while2(I,J,N,M, N2,M2) :-
    J =< M,
    p(I,J,N,M, I1,J1,N1,M1),
    J2 = J1+1,
    while2(I1,J2,N1,M1, N2,M2).
p(I,J,N,M, I1,J1,N1,M1) :- ...

The loops while1 and while2 are mutually recursive predicates. The procedure call p(i,j) is handled as a big step within a small-step semantics.
Big-step vs. small-step - block structure

**big-step**

\[
\text{block}(\text{St}0, \text{St}N) :- \\
\text{stmt1}(\text{St}0, \text{St}1), \\
\text{stmt2}(\text{St}1, \text{St}2), \\
\cdots \\
\text{stmtn}(\text{St}N-1, \text{St}N).
\]

\[
\text{stmt1}(\text{St}0, \text{St}1) :- \\
\cdots \\
\text{stmtn}(\text{St}0, \text{St}1) :- \\
\cdots
\]

**small-step**

\[
\text{block}(\text{St}, \text{St}Z) :- \\
\phi, \text{stmt1}(\text{St}1, \text{St}Z). \\
\text{stm1}(\text{St}0, \text{St}Z) :- \\
\phi, \text{stmt2}(\text{St}1, \text{St}Z). \\
\cdots \\
\text{stmtn-1}(\text{St}, \text{St}Z) :- \\
\phi, \text{stmtn}(\text{St}1, \text{St}Z). \\
\text{stmtn}(\text{St}, \text{St}Z) :- \\
\text{block1}(\text{St}, \text{St}Z). \ (\text{call to next block})
\]
The big-step or small-step encodings are used to construct predicates expressing reachable states in a program execution. Assume that \( \langle s, \sigma \rangle \) is the initial statement and state.

- **Small-step:**

\[
\text{reach} \langle s, \sigma \rangle \quad \text{where} \quad \langle s, \sigma \rangle \text{ is the initial statement and state}
\]

\[
\langle s, \sigma \rangle \Rightarrow \langle s_1, \sigma_1 \rangle \quad \text{reach} \langle s, \sigma \rangle \\
\text{reach} \langle s_1, \sigma_1 \rangle
\]

- In the same way as before, we can specialise these rules wrt. a given program, by unfolding the transitions \( \langle s, \sigma \rangle \Rightarrow \langle s_1, \sigma_1 \rangle \) and flattening structure.
Reachability predicates - big-step

- Big-step: a reachability rule is constructed for each syntactic case. E.g.

\[
\text{reach } \langle s_1 ; s_2, \sigma \rangle \\
\frac{\langle s_1, \sigma \rangle}{\langle s_1, \sigma \rangle} \quad \langle s_2, \sigma' \rangle \\
\text{reach } \langle s_1 ; s_2, \sigma \rangle \\
\frac{\langle s_1, \sigma \rangle \Rightarrow \sigma'}{\langle s_2, \sigma' \rangle} \\
\text{reach } \langle \text{while } (b) s, \sigma \rangle \\
\frac{\text{reach } \langle s, \sigma \rangle}{\text{if } b \text{ is true in } \sigma} \\
\text{reach } \langle \text{while } (b) s, \sigma' \rangle \\
\frac{\langle s, \sigma \rangle \Rightarrow \sigma'}{\text{if } b \text{ is true in } \sigma}
\]
The approach to deriving a reachability relation is the same as analysing queries in a CLP program. The construction above is analogous to a "magic-set" or "query-answer" transformation for CLP programs. For each predicate $p$ in a CLP program, introduce a query predicate $p^q$. Then for a clause

$$A \leftarrow B_1, \ldots, B_n$$

we derive the query clauses (for $1 \leq j \leq n$)

$$B_j^q \leftarrow A^q, B_1, \ldots, B_{j-1}$$
Horn clauses of the form

\[ \text{false} \leftarrow \phi, A_1, \ldots, A_n \]

are called "integrity constraints". The predicate \textit{false} (sometimes called \textit{error} or \textit{unsafe}) is always interpreted as false.
We now consider the translation of assume and assert statements.

We assume that the translation and specialisation yields a reachability predicate $\text{reach}_i(X_1, \ldots, X_n)$ for each program point $i$ in the imperative program.

Suppose that $\text{assert } P$ appears just before program point $i$, where $P$ is a constraint on the variables in scope at that point. Then create an integrity constraint

$$
\text{false} \leftarrow \text{reach}_i(X_1, \ldots, X_n), \neg P
$$
Suppose that *assume* $P$ appears just before program point $i$, where $P$ is a constraint on the variables in scope at that point. Then replace each clause of the form

$$reach_i(X_1, \ldots, X_n) \leftarrow B$$

by the clause

$$reach_i(X_1, \ldots, X_n) \leftarrow P, B$$
reach1(P, N) :-
    P != 0.
reach1(P, N) :-
    N = N1-1, reach4(P, N1).
reach2(P, N) :-
    N >=0, reach1(P, N).
reach3(P, N) :-
    N = 0, reach2(P, N).
reach4(P, N) :-
    N != 0, reach2(P, N).
reach4(P, N) :-
    P = 0, reach3(P1, N).
false :- P=0, reach2(P, N).

assume p != 0.
while (n >= 0) do
    assert p != 0;
    if (n==0)
        p := 0;
    fi
    n := n-1;
od
Good translations to CLP

To be useful for analysis and verification, the CLP program should:

- be of the same size order as the original program,
- predicates correspond to program points,
- remove all the source program syntax,
- predicate arguments should be relevant to the corresponding program point.
Approaches based on the control flow graph (which could be obtained from the language compiler).

The CFG naturally corresponds to small-step semantics (each edge in the CFG is a small step).

Intermediate representations (e.g. LLVM) can mix small-steps with big-steps for procedure calls.

Work is needed to compute (a) the variables relevant to a block; (b) suitable renaming of variables (e.g. SSA form).
Other translation approaches - threading the abstract syntax tree

The abstract syntax tree can be "threaded" with control flow (e.g. using a textbook attribute grammar). This is another way of generating a control flow graph, from which clauses corresponding to small-step semantics can be generated.
Reduction of the number of predicate arguments can be achieved by standard CLP techniques such as redundant argument filtering (Leuschel and Jørgensen 1996).

This is analogous to a live variable analysis.

The analysis can be performed forwards or backwards.
Redundant argument filtering - example

A fragment of a large example, filtered wrt. the integrity constraints.

Before filtering

After filtering

```
h1(E,H,K,A1) :-
  1=1.

h2(T1,W1,Z1,G2,P2) :-
  T1 = -(1+G2),
  h1(T1,W1,Z1,P2).

h3(T1,W1,Z1,G2,P2) :-
  h2(T1,W1,Z1,G2,P2).

h4(T1,W1,Z1,G2,P2) :-
  T1 >= 3,
  h3(T1,W1,Z1,G2,P2).

h5(T1,W1,Z1,G2,P2,R2,S2,T2,U2) :-
  h4(T1,W1,Z1,G2,P2),
  T1 = U2,
  S2 = 1 + U2,
  R2 = 1,
  T2 = 1.
```
CLP programs (should be) derived systematically from semantics:

- CLP representation of semantic judgements (e.g. operational semantics, proof rules)
- Semantics possibly instrumented or enhanced with traces, etc.
- Partially evaluate semantics wrt a fixed program to get a CLP program; obtain reachability predicates.
- Filter out the syntactic structures from the interpreter, leaving a CLP program over the domain of the program. Filter redundant arguments.

An excellent recent paper discussing many of these issues is the following:

*Semantics-based generation of verification conditions by program specialization*, De Angelis, Fioravanti, Pettorossi and Proietti, PPDP 2015.
The verification problem

- Let $P$ be a set of clauses expressing reachability, augmented with the integrity constraints and the assume conditions.
- $P$ is called the verification conditions.
- If $P$ has a model (is satisfiable) then the original program is safe wrt. to its assertions.
Equivalent formulations of the verification problem for Horn clauses

- $P$ has a model
- $P \not\models false$
- $false \not\in M[P]$ where $M[P]$ is the minimal model
- There is no derivation of $false$ from $P$.

These are of course undecidable problems. Arguably these equivalent formulations are the basis of the advantages of a CLP-based approach.
Main techniques for solving the verification problem

- Approximate the minimal model of \( P \) (using abstract interpretation) and check whether \textit{false} is present.
- Successively transform \( P \), preserving the minimal model, until \textit{false} is explicitly present or absent.
- Specialise \( P \) wrt derivations of \textit{false}, and apply one of the above methods.
- Successively approximate the minimal model of \( P \), applying counterexample-guided abstraction refinement (CEGAR) to improve precision.
Computing (approximate) models of CLP programs

SOURCE
LINEAR HYBRID AUTOMATA
IMPERATIVE PROGRAMS
HARDWARE

TRANSFORM to CLP
CLP interpreter + partial evaluation

TRANSFORM CLP PROGRAM
CONSTRAINT LOGIC PROGRAM

COMPUTE APPROXIMATE MODELS

M[P0]
M[P1]
Mq[P2]
Mq[Pn]

P0
P1
P2
Pn

+ Petri nets, assembly code, bytecode, functional programs, O-O languages, Z, B, logic programs, ...
A model is represented by a set of constrained facts.
The "immediate consequence" operator for a CLP program (a generalisation of the standard $T_P$ function).

$$T_P^C(I) = \begin{cases} 
A \leftarrow C \\
A \leftarrow B_1, \ldots , B_n, D \in P \\
\{ A_1 \leftarrow C_1, \ldots , A_n \leftarrow C_n \} \in I \\
\exists \theta \text{ such that } mgu((B_1, \ldots , B_n), (A_1, \ldots , A_n)) = \theta \\
C' = \bigcup_{i=1,\ldots,n} \{C_i\theta\} \cup D \\
\text{SAT}(C') \\
C = \text{proj}_{\text{Var}(A)}(C') \\
\end{cases}$$

$$M^C[P] = \text{lfp}(T_P^C)$$
The minimal model is computed as the least fixed point of the immediate consequences function $T_P^C$.

This is the limit of the Kleene sequence

$$\emptyset, T_P^C(\emptyset), T_P^C(T_P^C(\emptyset)), \ldots.$$ 

In general this is not a finite sequence – hence approximation is required.

- either in the model computation (bottom-up) or in the computation (top-down).
Abstract interpretation of CLP in one picture.

Safety condition: $T \circ \gamma \subseteq \gamma \circ S$
Abstraction by intervals, octagons, convex polyhedra, ...

\[
\begin{align*}
p_1(I, J) & : - \\
& I = 0, \ J = 10. \\
p_2(I, J) & : - \\
& p_1(I, J). \\
p_2(I, J) & : - \\
& I_1 \leq J_1, \\
& I = I_1 + 2, \\
& J = J_1 - 1, \\
& p_2(I_1, J_1). \\
p_3(I, J) & : - \\
& I \geq J + 1, \\
& p_2(I, J). \\
p_1(A, B) & : - \\
& 1 \cdot A = 0, \ 1 \cdot B = 10. \\
p_2(A, B) & : - \\
& 3 \cdot B \geq 17, \ -1 \cdot B \geq -10, \ 1 \cdot A + 2 \cdot B = 20. \\
p_3(A, B) & : - \\
& -3 \cdot A \geq -26, \ 3 \cdot A \geq 22, \ 1 \cdot A + 2 \cdot B = 20. \\
\end{align*}
\]

A sample convex polyhedral approximation. Advanced techniques (such as widening with thresholds) have been used to enhance precision.
Abstract interpretation provides a systematic framework for generating sound approximations of the models of CLP programs.

A great variety of useful symbolic and numeric abstract domains has been developed.

Abstraction can be combined with refinement heuristics to improve the precision of abstractions.

Generic optimisation of fixpoint computation has been studied (program SCCs, worklists, semi-naive evaluations, path focussing,...).
Proof by CLP transformation

John P. Gallagher
Analysis and verification of imperative programs through CLP
Proof by CLP transformation (overall idea)

Given a CLP program $P_0$, say we wish to show that some atom $A$ is not a consequence.

$$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k$$

$P_k$ contains no clause with head $A$

Suppose we wish to prove that $A$ is a consequence.

$$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k$$

$P_k$ contains a clause $A :- \text{true}$

Transformation rules preserve the model (wrt to some specified predicates).
The MAP system is an automatic program transformation system that automatically proves properties of CLP programs.

- Compares favourably with ARMC, HSF(C) and TRACER (see De Angelis et al. Sci. Comp. Prog. 2013)

Abstraction techniques related to abstract interpretation are used during the transformations.
The PRO-B system (Leuschel)

- The Pro-B system is an automatic program specialisation system that automatically proves properties of CLP programs.
- It is now being commercialised.
- The main proof technique is program specialisation - again aiming to make program properties explicit.
Model-preserving transformations are applied.
Proof is obtained when the required property becomes explicit in the transformed program.
Specialisation wrt a query is a very useful form of transformation – achieved by query-answer transforms, or by various specialisation algorithms.
Refinement techniques

- A given abstraction might not be precise enough to prove an assertion.
- An "abstract proof" of false might be obtained; however a concrete check of this proof might show that it is infeasible.
- The infeasible proof is examined, to find out where the current abstraction is losing precision.
- Either: a more precise abstract domain is developed based on this information; Or, the infeasible trace (and possibly a class of related traces) is physically eliminated by transforming the program.
- A key technique applied is Craig interpolation, to generalise the constraints appearing in the infeasible proof.
Some current challenges

- Semantics-based translation for full languages (e.g. the K-Framework of G. Roşu).
- Representation of memory/heap (e.g. separation logic "constraints", tree automata abstractions, ...)
- Verification of concurrent programs (see techniques by Grebenshchikov et al (PLDI 2012)).
- Refinement techniques for convex polyhedra and other domains.
Some Horn-clause-based tools for imperative program and verification

- MAP (Proietti et al) - iterated specialisation
- QARMC (Rybalchenko) - abstraction refinement with interpolation
- Eldarica (Kuncak et al) - abstraction refinement with interpolation
- SeaHorn (Gurfinkel) - abstraction refinement with interpolation
- CHA toolkit (Gallagher and Kafle) - convex polyhedral approximation, refinement by trace elimination.
- Pro-B (Leuschel) - specialisation.
- COSTA (Albert et al) - termination analysis, resource analysis
Acknowledgements

Thanks for collaboration and discussions with Maurizio Proietti, Alberto Pettorossi, Emanuele De Angelis, Fabio Fioravanti, Pierre Ganty, Manuel Hermenegildo, Pedro López, Bishoksan Kafle, Julio Peralta, Huseyin Saglam, Jorge Navas, Mike Codish, Michael Leuschel, David Monniaux, Arie Gurfinkel and many others.