Cref Documentation and User Manual

Mihaela Sighireanu

1 General Presentation

Cref is a static analyzer for programs using dynamic memory. It does mainly two kids of analysis:

check: The pre/post-condition reasoning on programs annotated with assertions into a first order logic L speaking about heap structure and content [**?**]. The analyzer is based on a satisfiability procedure for the logic L which is translated into a satisfiability problem for theories dealt by the current SMT solvers.

invgen: The invariant synthesis for "`while`" loops working on heap. The principle here is the computation of a fix-point over an abstract domain representing heaps. The synthesized invariants have a special syntactic form, but they express both properties on the structure of the heap and the data stored in the cells of the heap.

Architecture (*figure required*):

- From annotated C programs to Why we use e.g., Frama-C or Caduceous.
- From annotated Why programs to internal input we use Why and Cref.
- From Cref programs to Interproc programs (with pointer variables translated into special integer variables).
- The Interproc is called for the computation of the fix-point (library Fixpoint) on abstract domains for heaps (managed using the Apron¡ interface). One such abstract domain is Sll. It represents heaps for singly linked lists with data over some numerical domain.

From C to Why: The C programs can be annotated with special comments defining:

- vocabulary of the logic used to specify pre/post-condition and
- pre/post conditions

An existing solution to support such annotations is the Frama-C tool (based on old Caduceos) combined with the Jessie tool.

The current versions of these tools give a semantics for pointers to records which is not compatible with what we need. Some compilation work shall be done here.

From Why to Cref: This step is introduced only to provide a solution to the problem above. We generate an intermediate program on which operations on pointers are rewritten in the good semantics. Also, simple "`while`" loops are retrieved from the infinite loops of Why.

This phase shall disappear when a direct compilation using Frama-C is done.

From Cref to Interproc: This step is also a temporary glue used to facilitate the integration of the fix-point computation done in Fixpoint.

Since Interproc input language has only numerical variables, we translate pointer variables into numerical variables with special names as follows:

- if `v` is a pointer variable, it becomes `ptr_v` integer variable;
- `null` pointer becomes `ptr_null` integer variable;
- if `v.f` is a pointer field access, it becomes `ptr_v_ptr_f` integer variable;
- if `v.f` is a data field access, it becomes `ptr_v_dt_f` integer variable; we also introduce a new variable `ptr_v_dtp_f` in order to keep track of the old values of data fields.

The translation above is not done for a direct interface with Fixpoint. This interface require to rewrite a translation into a hyper-graph accepted by Fixpoint.

The atomic statements and conditions on pointer variables are translated in order to obtain a *normalized form* defined as follows:

- atomic boolean conditions on pointer variables can be only in the form: `v == u`, `v == null`, `v.f == u`, or `v.f == null`;
- assignment on pointer variables belongs to: `v = null`, `v = u`, `v = u.f`, `v.f = null`, or `v.f = u` (only if `v.f` has been put to `null` before).

This normalized form allows to deal with garbage collection semantics for heaps.

During this translation, the pre-conditions of the code analyzed are translated into elements of the abstract domain and are given as starting points to the main engine.

Main engine: The core part is the use of the fix-point computation on the hyper-graph of the program in conjunction with an abstract domain for heaps. The abstract domain used are managed using the Apron interface.

A External Libraries and Tools

A.1 Interproc

Interproc is a static analyzer of programs with numerical variables based on abstract interpretation. For this, it uses mainly two engines:

- Apron library for abstract representation of configurations over the numerical variables, and
- Fixpoint module for computation of fix-points of transformers (relations) given by a hyper-graph of elementary transformers.

The steps of Interproc are the following:

1. Parsing of files giving programs in a little imperative programming language with only scalar types (boolean, integer, float) and (recursive) procedures calls (files `spl_???`—parsing and AST).
2. Compute an hyper-graph of the program on which nodes are program control points and hyper-edges are blocks of statements of the program (files `syn2equation`, `equation`, `boolexpr`).
3. Call the Fixpoint computation forwardly/backwardly on the abstract domain chosen by the user (and managed by Apron) (file `solving` calling modules `Fixpoint` and `Apron`).

The remaining files are used for pretty-printing programs (`pSpl_syn`) and graphs.

The lacks observed during the use of Interproc:

- Interproc input language does not include an `assert` statement. Possible semantics for an assert statement in presence of fix-point computation: (1) stop the computation if the assertion is not satisfied, or (2) take as post-condition the assertion.

A.2 Apron

The lacks observed during the use of Apron:

- There is no other mean than syntactic restriction on names of variables to add attributes to the numerical variables. For example, integer variables which are symbolic parameters are dealt differently in the PDBM domain than the normal integer variables.

  It would be interesting to have an additional attribute for the type of each variable; this attribute is interpreted or not by each abstract domain.