Is Logic Effective for Analyzing C Programs?

Thomas Hillenbrand, Andreas Podelski and Dalibor Topić

Max-Planck-Institut für Informatik Saarbrücken
{hillen,podelski,topic}@mpi-sb.mpg.de

1 Introduction

We consider C programs that manipulate lists (*the* data structure to represent symbolic information) through operations defined in separate modules; operations such as `cons`, `nil`, `car`, `cdr`, `isNil`. We want to analyze such programs in order to detect simple programming errors (see the example given below). It turns out (see [CP98]) that model checking temporal properties can be reduced to *set-based program analysis*, as studied by Reynolds, Jones/Muchnik, Heintze, Vardi and others [Rey69,JM79,Hei92,FSVY91].

In the work described here [Top02], we perform the set-based analysis of list-processing C programs by

- translating a C program into a set of Horn clauses (i. e. a logic program) with the same operational semantics,
- applying an abstracting transformation (proposed by Vardi and others in [FSVY91]), thus obtaining a *uniform* logic program with Horn clauses of a restricted form; see (1) below,
- applying resolution to obtain Horn clauses of an even more restricted form; these Horn clauses are a declarative version of transition rules for tree automata.

Jones and Muchnik [JM79] presented the first set-based analysis of list-processing while programs; their automata-theoretic algorithm is based on Büchi's construction for *canonical rewrite systems* (which correspond to finite automata extended with push operations). Vardi and others [FSVY91] were the first to express abstraction as a source-to-source transformation of logic programs. Their algorithm is again automata-theoretic, based on the classical transformation of two-way to one-way automata.

To our knowledge, our work is the first that formulates, and implements, the analysis algorithm as a procedure on Horn clauses as a first-class data structure. We use a saturation-based theorem prover Spass [Wei97] as our programming system. One goal of our work is to evaluate the suitability of a general-purpose first-order theorem prover as the programming platform for program analysis tools. Furthermore, this work opens the avenue of research where we modify the 'set-based' abstraction, making it more and more precise. Instead of uniform logic programs, we may then obtain classes of logic programs for which the resolution procedure is *not* guaranteed to terminate.

We hope to be able to apply our experience in first-order theorem proving to the analysis problem and develop methods that terminate often enough in practice.

2 A sketch of the method employed

We follow the approach of [CP98]. That paper presents a method for the abstract verification of CTL properties of transition systems the states of which are modeled as ground atoms of logic programs. We explain the basic ideas by a simple example. The list operators `cons`, `nil`, `car`, `cdr` shall be implemented with the usual semantics. The following fragment is an attempt to list reversal; the assignment to the variable `x` within the while loop leads to non-termination whenever the initial value of `x` is not the empty list.

```
    y = nil();
p   while (!isNil(x)) {
      y = cons(car(x),y);
      x = cons(cdr(x),nil());
   }
q   x = y;
r
```

As a prerequisite, the C program to be analyzed needs to be filtered with respect to the aspects that are relevant to our analysis. That is, all the expressions not being related to the list operations, like arithmetic calculations or character string manipulations, are replaced by constants. For the example at hand, nothing needs to be done.

The syntax tree is then used to convert the filtered imperative program into an equivalent logic one. For every program location there is a corresponding predicate; and the arguments of the predicate correspond to the variables visible at that location. Transitions between program locations are reflected by implications between corresponding predicate instances. Conditions in the transitions are expressed by the patterns of the clause heads. The above code is translated into the following logic program:

$$
\begin{aligned}
\mathit{init}(U) &\leftarrow p(U,[\,]) \\
p([U|U'],V) &\leftarrow p([U'],[U|V]) \\
p([\,],V) &\leftarrow q([\,],V) \\
q(U,V) &\leftarrow r(V,V)
\end{aligned}
$$

For example, the second and the third clause reflect that in program location $\boxed{\text{p}}$, if the variable `x` holds a non-empty list, then its head is put in front of the list `y` and `x` becomes the one-element list consisting of the former tail, whereas in case `x` is empty the program execution continues at location $\boxed{\text{q}}$.

In the next stage, a set-based abstraction of the logic program is computed. The abstraction is, in the spirit of [FSVY91], again expressed as a logic program, and can be considered as inferring types for the program predicates. For every (wlog. monadic) Horn clause

$$p(t) \leftarrow \text{body}$$

where t is a linear term with variables $X_1, \ldots, X_n$, we introduce n new predicates $p_1, \ldots, p_n$ and translate the Horn clause into

$$\begin{aligned} p(t) &\leftarrow p_1(X_1), \ldots, p_n(X_n) \\ p_1(X_1) &\leftarrow \text{body} \\ &\vdots \\ p_n(X_n) &\leftarrow \text{body} \end{aligned}$$

Finally the resulting clause set is transformed such that every element has one of the following forms:

$$\begin{aligned} p(f(X_1, \ldots, X_n)) &\leftarrow p_1(X_1) \wedge \ldots \wedge p_n(X_n) \\ p(X) &\leftarrow p_1(X) \wedge \ldots \wedge p_m(X) \\ p(X) &\leftarrow p'(t) \end{aligned} \tag{1}$$

The least and the greatest model of the abstracted logic program are supersets of the corresponding models of the initial program. This relationship allows us to conservatively approximate the application of CTL properties by corresponding fixpoint operators. The models of the abstract program can be represented by non-deterministic tree automata. Alternatively, resolution reasoning on Horn clauses can be employed to obtain such a representation. A clause $p(f(X,Y)) \leftarrow q(X), r(Y)$ corresponds to a non-deterministic transition from state p, where the tree root is labeled with f, to states q and r dealing with the left resp. right subtree. The saturation process always terminates on the logic fragment at hand. We will use the SPASS theorem prover [Wei97] as an implementation of saturation [BG94].

References

[BG94] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[CP98] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 358–378. Springer-Verlag, 1998.

[FSVY91] T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In A. R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. IEEE Computer Society Press, 1991.

[Hei92] N. Heintze. *Set-based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pages 244–256. ACM Press, 1979.

[Rey69] J. C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[Top02] D. Topić. Static analysis of list-processing C programs. Diplomarbeit, Universität des Saarlandes and Max-Planck-Institut für Informatik, 2002. In preparation.

[Wei97] Chr. Weidenbach. SPASS version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.