Interfacing Dynamically Typed Languages and the Why Tool: Reasoning about Lists and Tuples

Cláudio Amaral [*]
Chalmers University of Technology
LIACC, University of Porto
amaral@chalmers.se

Mário Florido
DCC-FC, University of Porto
LIACC, University of Porto
amf@ncc.up.pt

Patrik Jansson
Chalmers University of Technology
patrik.jansson@chalmers.se

Abstract

Formal software verification is currently contributing to new generations of software systems that are proved to follow a given specification. Unfortunately, most dynamically typed languages lack the tools for such reasoning.

We present a tool used to help verify some user specified properties on a small language. The process is based on functional contracts with annotations on the source code that later are transformed into logic goals that need to be proved in order to conclude that the program meets its specification. As part of the tool we also present a term model for dynamically typed data structures.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Languages, Verification

Keywords Erlang, Why tool, Verification Condition

1. Introduction

To abstract from programming language and provers, similarly to compilers, Program Verification tools increasingly rely on Intermediate (Verification) Languages. These languages are designed to be closer to the logics of the theorem provers for an easier translation. They are a great tool to avoid error prone effort duplication and have good support for many programming language/logic features. Dynamic typing is a key feature in many interesting and widely used programming languages, such as Erlang [1].

However, reasoning about dynamically typed values is difficult because it is harder to know their sorts than in statically typed languages. This difficulty manifests by the lack of direct support for dynamically typed languages. An example of this is the Why tool [2], with front-ends for C and Java [3–5]. Why is a verification condition generator (VCG) for a small intermediate language (let us call it the *Why Language*, WL for short) with annotations, which enable property specification, and type safety, through simple type checking.

The general goal in this paper is to present a framework[1] to support verification of dynamically typed languages by defining basic support in WL for dynamically typed structures, using logic definitions and axioms, and to translate a small dynamically typed functional language with Erlang-like syntax to WL. We can then express and verify properties of functions and values in a small pure and sequential subset of Erlang programs.

Section 2 presents some key aspects of our model for the translated values in WL. Section 3 shows with an example the framework use. Finally, Section 4 gives an overview of the tool elements and steps.

[*] author is funded by FCT grant SFRH/BD/65371/2009; corresponding author

Erlang'11, September 23, 2011, Tokyo, Japan.
ACM 978-1-4503-0859-5/11/09.

2. Modelling terms

In a dynamically typed language, a given program data value has a set of possible types or kinds of values until the execution flow reaches it. This set is usually a composition of known different elements. Lets define the `erl_term` type as the type of all values of our programs after translation to WL. It represents all kinds of values, which in our case are lists, tuples, atoms and integers.

The sub-types of `erl_term` are represented in WL as first-order logic functional symbols. They are available in the program through an axiomatic semantics. For each sub-type type there are wrapping/unwrapping dual functional symbols to/from `erl_term`, with the exception of atoms which are represented as term constants. We model lists by the type `erl_list`, together with the logic functional symbols to wrap/unwrap lists, and necessary axioms.

```
type erl_list
logic term_list : erl_list -> erl_term
logic get_list  : erl_term -> erl_list
```

The first is an abstract type declaration and the others are declarations of logic functional symbols. In these last declarations there is a type signature (after the ':'), representing the type of the functional symbol when applied to the specified number of arguments with according type.

Lists are data structures that consist of a possibly empty sequence of data records, called *elements* or *nodes*. This implies the existence of empty and non-empty lists. Non-empty lists have at least one element, the first node or *head*, and the rest of the list is called the *tail*.

```
(* list construction *)
logic nil  : erl_list
logic cons : erl_term, erl_list -> erl_list
```

[1] Code is available from `http://www.cse.chalmers.se/~amaral/erl2why.html`.

```
(* data access *)
logic head : erl_list -> erl_term
logic tail : erl_list -> erl_list
(* axioms... *)
```

To complete the basic support for lists we also provide some test functions on lists' structure and organisation, giving the programmer the means to differentiate list values. Among the tests are the empty-list test, one of the most useful, and whether or not the concrete value of a term is of type `erl_list`. Length and element positioning are also part of the framework.

```
logic is_nil : erl_list -> bool
logic is_list : erl_term -> bool
(* axioms... *)
(* length *)
logic list_length: erl_list -> int
(* position access *)
logic nth : int, erl_list -> erl_term
(* axioms... *)
```

Although tuples differ conceptually from lists, their semantics as an ordered sequence of elements is similar. Therefore, we have used the list model to build the tuple model. As with lists, we have defined the basic functionality for reasoning about and manipulating tuples, whose details are omitted here. The target for the source language integers are WL integers, encapsulated by the wrapper `erl_term`, in same way as list and tuple values are (`term_integer(42)`).

3. Example

To get a better understanding of our goal, consider the following piece of annotated code. In this example we have an append-free version of list reverse.

```
%@ requires:
%    is_list(Arg1) = true
reverse(L) ->  reverse_aux(L,[]).
%@ ensures:
%     erl_list_length(get_list(Arg1))
%       = erl_list_length(get_list(result))
%   and (forall i:int.
%     1<=i<=erl_list_length(get_list(L))   ->
%      nth(i,get_list(L))
%        = nth(erl_list_length(get_list(L))+1-i,
%               get_list(result)))
```

In order to specify the intended behaviour of this code, we need to provide proper annotations referring to the properties we want to reason about. They are identified by `requires`/`ensures` labels in comments starting with the @ symbol, which is a conventional notation used in some specification languages [6, 7].

This `reverse` implementation is only designed to receive a list. This is specified by requiring that the first argument passes the `is_list` test.

One of the expected outcomes is that the returned value is a list of the same length. This is easy to describe for `reverse`, simple equality between list lengths. It is important to see that, by using the `get_list` logic function on the result, if the post-condition can be proved correct it is implied that the result is a list because of its type. The other expected property of the result is the reversed order of the elements in the returned list, described by the *forall* formula in the post condition.

To verify `reverse` we also need a specification of `reverse_aux` behaviour with respect to the values of the calling arguments.

```
%@ requires:
%    is_list(Arg1) = true
%   and is_list(Arg2) = true
reverse_aux( [], LAcc ) ->
    LAcc;
reverse_aux( [H|RL], LAcc ) ->
    reverse_aux(RL,[H|LAcc]).
%@ ensures:
%    erl_list_length(get_list(result)) =
%      erl_list_length(get_list(Arg1))
%      + erl_list_length(get_list(Arg2))
%  and (forall i:int.
%   ( 1<=i<=erl_list_length(get_list(Arg1)) ) ->
%    ( nth(i,get_list(Arg1)) =
%        nth(erl_list_length(get_list(Arg1))+1-i,
%            get_list(result))))
%  and (forall j:int.
%   ( 1<=j<=erl_list_length(get_list(Arg2)) ) ->
%    ( nth(j,get_list(Arg2)) =
%        nth(erl_list_length(get_list(Arg1))+j,
%            get_list(result))))
```

This behaviour will depend on both arguments, relating the length of the result with the length of the arguments, as well as positions of key elements of the lists in each call (invariant). With this example, all but one of the verification conditions (VCs) generated by Why are possible to check automatically. Even with such specifications, the most interesting property (reversed positioning in the result list) had to be discharged using an interactive prover, although it was relatively simple.

4. Tool overview

The tool is composed by a compiler to the WL language and some WL libraries with the models for the translated values. The input of the tool is a valid Erlang module. Only a small subset of the language is actually supported (there is no concurrency nor module system, for example). The output is a WL version of the input module that is passed to the Why tool together with our libraries to produce the VCs for the given program.

The input file will have contracts for the functions and possibly some other specification elements, through dedicated attribute forms. These elements are only for specification purposes, they do not interfere with the program execution.

The properties, and formulas in general, in such specifications are first-order logic formulas. The properties must be written according to the models of the values in the tool's WL libraries.

The resulting WL file may then be used in any way Why files can and the generated VCs proved with any of the provers installed and supported by Why, automatic or interactive.

The resulting WL file may then be used in the Why tool. The generated VCs can be proved with any of the supported provers.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN 193435600X, 9781934356005.

[2] J.-C. Filliâtre. Why: A Multi-language Multi-prover Verification Tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[3] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, Nov. 2004. Springer-Verlag.

[4] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for Certification of JAVA/JAVACARD programs annotated in JML. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.

[5] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual*, boron edition, April 2010.

[6] G. T. Leavens, E. Poll, C. C. Y. Cheon, C. Ruby, D. Cok, P. MÃijller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, draft edition, June 2008.

[7] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010.