SAML: Static Analysis in Standard ML

Haakon Larsen

Introduction

SAML is a framework to be used for static analysis, and is written in Standard ML (SML), a statically typed functional language. In its complete form, SAML is intended to provide a framework that will facilitate the formulation of analyses on a program's AST and a set of key such examples.

Goals

The overarching goal is to take advantage of SML's expressive power to create a highly modular base system that allows for certain checks (especially ones that checks for possible "unsafe" paths in a program's AST) to be formulated across a set of languages (e.g. C, Ada, Java). The stated goal will be accomplished by furnishing parsers and translators from the respective languages into a specified meta-language on which the checks can be performed. Checks written in the meta-language are therefore valid for each language in the supported set, given that the translations are correct. The initial intent is for many of these tests to be expressed using a folding system, which will be described below.

The goal for the present semester was to design and implement a proof-of-concept of the folding system for the C language and investigate whether it can be used to specify useful checks on a program's AST. Specifically, I wrote a prototype that uses the C language folding system to check whether there are possible paths from a pointer's declaration to it's reference that do not include a check to see whether the pointer is null (i.e. there is a *possible* null-pointer reference).

Folding System

Folding is an elegant and powerful functional programming concept used to traverse data structures. As an example, parametrically typed lists are included in the SML/NJ[1] distribution and the List library comes with a function that folds over SML's lists. As one can traverse a list in either left-to-right or right-to-left order, there are two such functions, but they both have the following type signature:

```
val foldl : (('a * 'b) -> 'b) -> 'b -> 'a list -> 'b
```

What this signature means is that foldl (left-to-right folding) takes in three arguments, which will be described below. The last argument ('a list) says that foldl will operate on a 'a list, where the 'a is some arbitrary type (i.e. a type variable). The second to last argument (of type 'b) is the initial accumulator, which is returned if the list is empty. The

[1] SML of New Jersey – the variant of SML used for SAML

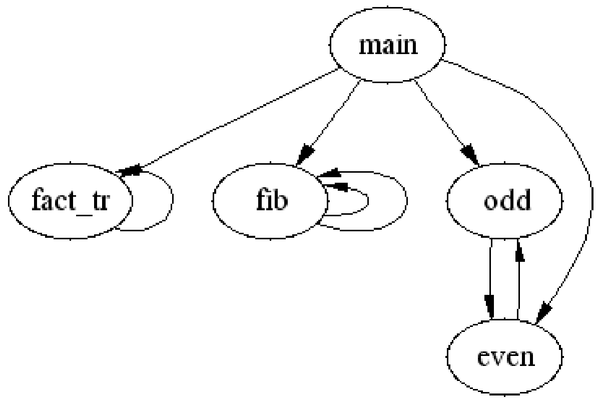
first argument is a function (‘a * ‘b -> ‘b) that is called for every ‘a element of the ‘a list. The function is also passed in the current accumulator (of type ‘b), and the result of the function becomes the accumulator for the next iteration. Once all elements of the list have been visited, the final accumulator is returned as the result[2].

Folding is extremely powerful because it allows the user to specify a function (hereafter called the folding function) that only specifies the new accumulator given the current element and current accumulator, along with a base-case accumulator. Nearly every other function specified in the SML List library could be implemented in a purely functional manner using foldl.

Against this backdrop, therefore, the first two-thirds of this semester was spent writing a folding system for C. Ultimately, as noted, the goal is for SAML to provide a meta-language for certain checks to be performed on ASTs over a set of languages (and hence a folding system written for this meta-language). The present version, however, is a proof-of-concept intended to illustrate that this is, at least, seemingly feasible. The folding is presently done in pre-order and is essentially[3] a mirror of List.foldl as implemented over ASTs.

Callgraph

A callgraph is a graph where the nodes are functions and directed edges go from the caller to the called. The callgraph module is a simple prototype module that can produce the callgraph of an AST. The purpose is merely to show that the folding system was in place and to hint at its expressiveness (the module can produce graphs of the nature bellow in a meager 80 lines). For completeness sake, a sample generated callgraph is included below.



[4]

[2] The description above is a slight simplification given that foldl is a higher-order function using currying, but this is not relevant to explaining folding itself
[3] ‘Essentially’ because the folding system is written using higher-order structures (i.e. functors), which allows for greater modularity
[4] The graphs presented in this report are generated using ATT’s dot language (http://www.research.att.com/~erg/graphviz/info/lang.html) and dot graph generator (http://www.research.att.com/sw/tools/graphviz/)

Null-paths

The null-paths module is a prototype that creates a graph representing key-events in the life of a pointer, from the perspective of trying to identify paths where references of this variable might be made without having first verified that the pointer is not null (i.e. possible null-pointer references).

The algorithm is determined entirely by the folding function and the initial accumulator passed into the C folding system described above[5]. The key component of the algorithm is the graph that represents the important events in a pointer's lifespan. Nodes in the graph are therefore creation, use, check (for whether it is null), call-site, and formal parameter specification – all of which are associated with (at least) an identifier (representing the C pointer). The second major component of the algorithm is the mapping from a pointer to its most recent (important) event – this is what produces the edges between the numerous events.

In that it is a prototype, there are numerous complications that are not considered by the module. It can, for example, only handle the simplest check for a pointer not being null, and it does not account for pass-by-reference. Yet I do believe that the prototype shows that interesting checks can be performed using the folding system.

As an example, the null-paths prototype was run on the C code below on the left to generate the graph on the right.

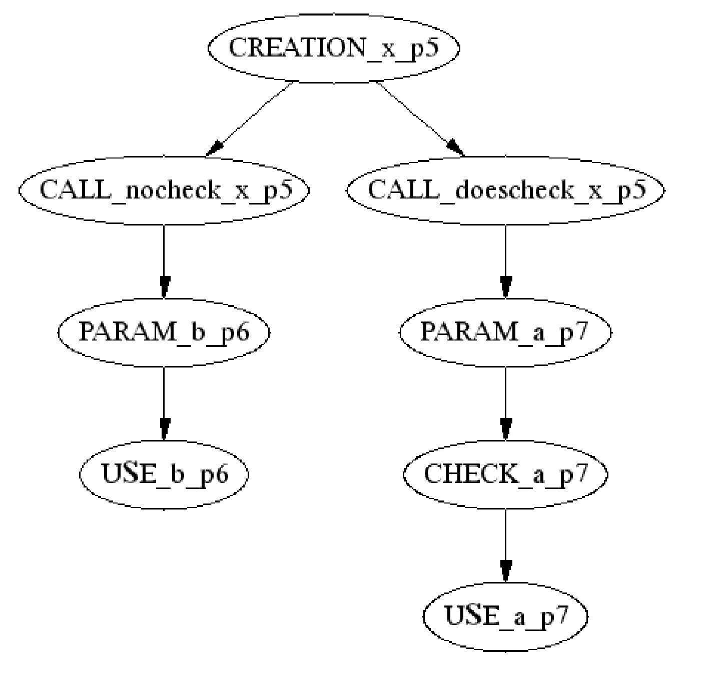
```
void* malloc(int);

int doescheck(int*);
int nocheck(int*);

int main () {
  int* x =
     malloc(sizeof(int));
  doescheck(x);
  return nocheck(x);
}

int nocheck(int* b) {
  return *b;
}

int doescheck(int* a) {
  if (a)
    return *a;
  else
    return -1;
}
```



[5] This determinism follows directly by the specification of a folding system (e.g. true for SML's foldl also) as long as the folding function is implemented using the functional programming paradigm (i.e. no references)

Conclusion

It is my belief that the null-paths prototype shows that the folding system has promise. The prototype highlights the expressiveness of the folding system in that it is written in little over 150 lines (excluding external data structures such as red black trees and graphs). While counting the lines of SML code can be severely misleading (on account SML's expressiveness itself), I do believe this is noteworthy. The null-paths module is, as noted, in its infancy and further investigation is required before certain knowledge of its usefulness can be determined. I intend to investigate the null-paths module further before delving into the feasibility of the discussed meta-language.