Documentation

Beerend Lauwers Hidde Verstoep Frank Wijmans

June 7, 2012

1 Introduction

We have analysed Javascript code using Attribute Grammars in Haskell. The purpose of our analysis is to find out about the possible typing of variables and functions, since Javascript is a dynamic and weakly typed language. This document provides information about the following source code files:

1. `Attributes.ag`

   Basic attributes are defined here.

2. `Datatypes.ag`

   Contains our AG datatypes based upon those of the parser.

3. `Flow.ag`

   The edges and vertices of the flow graph are built here.

4. `Scope.ag`

   Special attention for scoping, i.e. attending to the details of variables and functions.

5. `Transfer.ag`

   Transfer functions are defined here for use in the analysis algorithm. The algorithm is named `alg` in the source code.

6. `Convert.hs`

   The conversion between parser and the 'ag-datatypes'.

To use our analysis on a Javascript file, first you have to build the executable using the `make` command. Then, use the following command to generate a *.png/*.pdf file: `test.sh /folder/filename.js`. This will generate a PDF and a PNG file of the flow graph. For example, `test.sh /examples/ltgt.js` will generate the files `ltgt.dot.png` and `ltgt.dot.pdf`.

2 Documentation

2.1 Language

We have used the HJS Javascript library as found on Hackage. We noticed that not all datatypes defined in the library were actually used as a (partial) result

in the parsing process. We translated the generated AST to our own AG AST (defined in `Datatypes.ag`) in `Convert.hs`. As a result, our AG does not contain all the datatypes from the library.

2.2 Flow Graph

We generate an image of a directed flow graph, starting at `program-begin` and ending at `program-end`. Black edges simply follow evaluation order (as defined in [1]). Red edges denote interprocedural edges. Green edges point from a call program point to the return program point and are only included to improve readability.

2.3 Transfer functions

Transfer functions are generated for all edges of the flow graph, after which they are added to the work list. The work list algorithm then calculates all the types a certain program point may have. The algorithm and transfer function code is defined in `Transfer.ag`.

2.4 Lattice

Our lattice code can be found in `Transfer.ag`. We have chosen a lightweight implementation using a `Lattice` Haskell type class, of which instances were made for sets, where our `join` operator is set union and our `meet` operator is set intersection, and one for the Data.Map datatype, where `join` and `meet` are respectively `Data.Map.unionWith join` and `Data.Map.intersectionWith meet`; the `join` and `meet` functions are those of the values of the Map. We define a simple datatype type `Type` that contains the types we are able to infer. Our lattice is then defined as a set of primitive types: `type TLattice = Set Type`. We then define a type synonym for use in our transfer functions: `type Data = Map Vertex TLattice`. Our transfer functions go from `Data` $\rightarrow$ `Data`.

2.5 Design choices

Our analysis is not flow-sensitive or path-sensitive: all types of a variable are collected at the declaration program point, irrelevant to when or where the types are inferred. It is also not context-sensitive. In the following code snippet, both `x` and `y` will be inferred to have the types `TNumber` and `TString`, as those are the types the function `id` may generate during the entire program run.

```
1  function id (x) { return x; }
2  var x = 0;
3  var y = "test";
4  x = id(x);
5  y = id(y);
```

```
1
2  function lt (x,y) {
3    return x<y;
4  }
5
6  var gt = function (x,y) {
7    return x>y;
8  }
9
10 var x = 0;
11 var y = 10;
12
13 while(lt(x, 10)){
14
15   if(!gt(y, 1)){
16     y=y-2;
17   }
18   else{
19     x++;
20   if(x==10){y="done.";}else{y=10;}
21   }
22 }
```

Figure 1: Javascript code - Ltgt.js

3 Examples

3.1 Ltgt.js

The Javascript code in figure 1 has two functions and a while loop using these two functions. Notice the difference in defining the functions and the usage of the variable y.

To show how this is formulated in the analysis, let's walk through the generated graph and draw the links between code and result.

The function of the code snippet might be a little trivial, but a few language concepts are encapsulated within this example.

1. Variables that contain more than one type. In this example it is clear that the two following assignments will not infer the same type.

```
1     var y = 10;  \\ TNumber
2     y = "done."; \\ TString
```

2. Variable names are shadowed. We use `x` and `y` in the global scope, but we also shadow these inside the functions `gt` and `lt`. We even mix up `x` and `y` to make sure the scoping rules are visible.

3. Function declaration styles.

```
1           function lt (x,y) {...}   \\ TFunction
2  var gt = function    (x,y) {...};  \\ TFunction
```

   First up is the usual way of declaring functions, where the top level gets access to the function (in other words, it becomes globally accessible). Secondly we have an anonymous function declaration that we assign to a variable. These should hold no difference in the flow graph

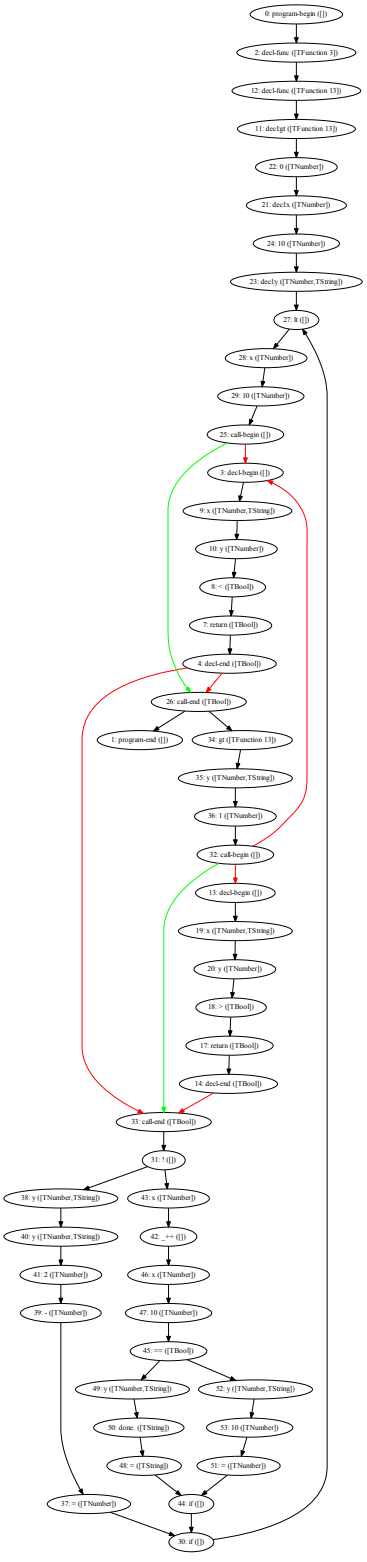

Figure 2: The graph image generated from Ltgt.js

When looking at the graph we see that it closely follows the code. From nodes 0 to 23 we find the declaration of two functions and the assignment of `gt`, as well as the initialization of `x` and `y`, and respectively their first assignments.

The call of `lt` is on node 25 on the values `x` and `10`. Continuing into the loop body, something interesting happens: on node 9 you see the first shadowing due to scoping, namely on the name `x`. Let us use `x'` to denote that it is a shadowed `x`. Node 23 states that `y` is a number and a string during the entire run of the program. As this top-level `y` is passed to the `gt` function in the first `if` construct, the types of `x'` become the same as those of `y`.

Having called `gt`, we take the negation of the resulting value. This is visible in node 31. Following the structure, the interesting part happens in the `if` construct inside the outer `else` branch. We see here that `y` is assigned either "done."(node 48) or 10 (node 51). Node 44 shows the end of the inner `if` construct, and node 30 does the same for the outer `if` construct, after which program flow goes back to node 27 to begin another iteration of the `while` loop.

4 Future Work

In order to accurately analyse objects, they would have to be evaluated. Fully evaluating the source code before passing it to do analysis is a logical extension of our work. Making our analysis context-sensitive is another worthwhile extension that can be looked into. We were unable to do this due to time constraints.

References

[1] E. International, *ECMA-262: ECMAScript Language Specification*, 5.1 ed., June 2011.