Assignment 3 - Semantic Analysis

Mayer Goldberg & Lior Zur-Lotan

December 13, 2021

Contents

1 General

- You may work on this assignment alone, or with a single partner. You may not join a group of two or more students to work on the assignment. If you are unable to find or maintain a partner with whom to work on the assignment, then you will work on it alone.
- You are not to copy/reuse code from other students, students of previous years, or code you found on the internet. If you do, you are guilty of *academic dishonesty*. All discovered cases of *academic dishonesty* will be forwarded to *disciplinary committee* (וַעֲדַת מִשְׁמַעַת) for disciplinary action. Please save yourself the trouble, and save us the effort and the inconvenience of dealing with you on this level. This is not how we prefer to relate to our students, but *academic dishonesty* is a real problem, and if necessary, we will pursue all disciplinary venues available to us.
- You should be very careful to test your work before you submit. Testing your work means that all the files you require are available and usable and are included in your submission.
- Make sure your code doesn't generate any unnecessary output: Forgotten debug statements, unnecessary print statements, etc., will result in your output being considered incorrect, and you will lose points. You will not get back these points by appealing or writing emails and complaints, so **please** be careful and make sure your code runs properly
- Your code should generate absolutely no warnings or error messages. If it does, you might get a grade of zero. No appeals shall be accepted in this case.
- Please read this document completely, from start to finish, before beginning work on the assignment.

2 Introduction

The purpose of this assignment is to add the *semantic analysis* component to the pipeline of your compiler. This component shall compute and annotate the *lexical addresses* of all variables, annotate applications in *tail-position*, and *box* variables that are copied from the stack to the heap, and that changes to which may be visible elsewhere in the code.

3 The semantic-analysis module

3.1 The `expr'` type

The following type declarations appear in the newly added `semantic-analyser.ml` file:

```
type var' =
  | VarFree of string
  | VarParam of string * int
  | VarBound of string * int * int;;

type expr' =
  | ScmConst' of sexpr
  | ScmVar' of var'
  | ScmBox' of var'
  | ScmBoxGet' of var'
  | ScmBoxSet' of var' * expr'
  | ScmIf' of expr' * expr' * expr'
  | ScmSeq' of expr' list
  | ScmSet' of var' * expr'
  | ScmDef' of var' * expr'
  | ScmOr' of expr' list
  | ScmLambdaSimple' of string list * expr'
  | ScmLambdaOpt' of string list * string * expr'
  | ScmApplic' of expr' * (expr' list)
  | ScmApplicTP' of expr' * (expr' list);;
```

The output of the semantic analysis phase of your compiler should be of type `expr'`, and this is what the code generator (in the final project) shall receive.

All the work on the semantic analyser should be placed in the `Semantics` module of `semantic-analyser.ml`. You are required to implement the following functions as described below: `annotate_lexical_addresses`, `annotate_tail_calls`, and `box_set`. The already implemented function `run_semantics` uses the above functions to run the full semantic analysis process. The `run_semantics` function enforces a calling order on the above functions: Firstly, `annotate_lexical_addresses` shall be called, the return value of which shall be passed as an argument to the procedure `annotate_tail_calls`, the return value of which shall be passed to the procedure `box_set`.

3.2 Lexical addressing

You shall provide an implementation for the procedure `annotate_lexical_addresses : expr → expr'` (as specified in the skeleton file `semantic-analyser.ml`), where all `ScmVar` records have been replaced by `ScmVar'` records. Notice that the type constructor `ScmVar'` holds a value of type

`var'`, which is a disjoint type made of `VarFree`, `VarParam`, and `VarBound`. So instances of `ScmVar'` should include their lexical address.

Lexical addressing was presented in class:

- Parameter instances should be tagged using the `VarParam` type constructor. The `string` value should be the repackaged variable name and the `int` value should be the minor index of the parameter in the closure (0-based index).
- Bound variable instances should be tagged using the `VarBound` type constructor. The `string` value should be the repackaged variable name, the first `int` value should be the major index of the bound instance in the closure (0-based index), and the second `int` value should be the minor index of the bound instance in the closure (0-based index).
- All variable instances which are neither parameter nor bound instances are free variable instances. Free variable instances should be tagged using the `VarFree` type constructor, in which the `string` value should be the repackaged variable name.

Notice that `annotate_lexical_addresses : expr → expr'`. The other operations, of annotating tail-calls, and boxing variables take arguments of type `expr'`, so `annotate_lexical_addresses` is the *first* procedure we need to run on the tag-parsed expression.

3.3 Annotating tail calls

In annotating tail-calls, your compiler will need to replace some instances of `ScmApplic'` with corresponding instances of `ScmApplicTP'`. You should go over each and every form in the type `expr'`, and call the procedure `annotate_tail_calls` on each of the sub-expressions, to make sure the entire AST is converted, all the way to the leaves.

3.4 Boxing of variables

In class, we presented two criteria for boxing variables:

- The variable has (at least) one *read* occurrence within some closure, and (at least) one *write* occurrence in *another* closure.
- Both occurrences and neither both parameters, nor already refer to the same rib in a lexical environment.

As we mentioned in class, these rules are always sufficient, but sometime unnecessary, i.e., there are some cases in which boxing is not logically required, but using these criteria will still lead to boxing. Consider the following example:

```
(define foo
  (lambda (x)
    (set! x 1)
    (lambda () x)))
```

Note that the order of evaluation of the expressions in the body of procedure `foo` is known statically: the body of the lambda expression contains an implicit sequence of two expressions, that are evaluated from first to last. This means that logically, `x` need not be boxed. However, since the two criteria presented in class do not consider evaluation order, using those rules would result in `x` being boxed.

3.4.1 How to implement boxing

For each variable `VarParam(v, minor)` that should be boxed, you must take the following three steps:

1. Add the expression `ScmSet'(VarParam(v, minor), ScmBox'(VarParam(v,minor)))` as the first expression in the sequence of the body of the `lambda`-expression in which it is defined.
2. Replace any *get*-occurances of `v` with `BoxGet'` records. These occurrences can be either parameter instances or bound instances.
3. Replace any *set*-occurrences of `v` with `BoxSet'` records. These occurrences can be either parameter instances or bound instances.

4 Submission

You are required to submit a link to your Git repository, as instructed in the README.md file in the repository.

4.1 Interface

The file `semantic-analyser.ml` is the interface file for your assignment. The definitions in this file will be used to test your code. If you make breaking changes to these definitions, we will be unable to test and grade your assignment. Do not break the interface. Operations which are considered interface-breaking:

- Modifying the line: `#use "tag-parser.ml"`
- Modifying the types defined in `"tag-parser.ml"`
- Modifying the types defined in `"reader.ml"`
- Modifying the module signatures and types defined in `"semantic-analyser.ml"`

Other than breaking the interface, you are allowed to add any code and/or files you like. Please note that any modifications you make to `pc.ml` and any file in the `tests` folder will be discarded during the testing process.

The patch you submit for this assignment includes your work from assignment 1 and 2, along with any corrections you made, in addition to your work on assignment 3. Despite this, only your work on this assignment (assignment 3) will be tested and graded. That means we will **not** run your reader to generate sexprs for the tag-parser, and we will **not** run your tag-parser to generate exprs.

4.2 Assignment Statemenmt

Among the files you are required to edit is the file `readme.txt`.

The file `readme.txt` should contain

1. The names and IDs of all the people who worked on this assignment. There should be either your own name, or your name and that of your partner. You may only have one partner for this assignment.

2. The following statement:

   I (We) assert that the work we submitted is 100% our own. We have not received any part from any other student in the class, nor have we give parts of it for use to others. Nor have we used code from other sources: Courses taught previously at this university, courses taught at other universities, various bits of code found on the internet, etc.

   We realize that should our code be found to contain code from other sources, that a formal case shall be opened against us with the *disciplinary committee* (וַעֲדַת מִשְׁמַעַת), in pursuit of disciplinary action.

5 Testing and grading your assignments

To test your assignment, refer to the **Testing your Semantic-Analyzer** Section in your Git repository.

Note that we will **not** test your semantic-analyser on illegal inputs, therefore, you are not required to provide meaningful outputs for such inputs.

6 A word of advice

The class is very large. We do not have the human resources to handle late submissions or late corrections from people who do not follow instructions. By contrast, it should take you very little effort to make sure your submission conforms to what we ask. If you fail to follow the instructions to the letter, you will not have another chance to submit the assignment: If files your work depends on are missing, if functions don't work as they are supposed to, if the statement asserting authenticity of your work is missing, if your work generates output that is not called for (e.g., because of leftover debug statements), etc., then you're going to have points deducted. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.

6.1 A final checklists

1. You completed the module skeleton provided in the `semantic-analyser.ml` file
2. `annotate_lexical_addresses`, `annotate_tail_calls`, and `box_set`. procedures match the behavior presented in class.
3. You did not change the module signatures
4. Your semantic analyser runs correctly under OCaml on the provided Linux image
5. You completed the readme.txt file that contains the following information:
   (a) Your name and ID
   (b) The name and ID of your partner for this assignment, assuming you worked with a partner.
   (c) A statement asserting that the code you are submitting is your own work, that you did not use code found on the internet or given to you by someone other than the teaching staff or your partner for the assignment.
6. You submitted the `submission` file with your repository link

7. You cloned a fresh copy of your repository and tested it as described in the README.md file under **Testing your Semantic-Analyzer**