Assignment 3 (Due: Tuesday, 18/12/18, 10:00AM)

Mayer Goldberg

December 3, 2018

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 *va'adat mishma'at* 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 get a grade of zero.
- No late submissions will be accepted, no exceptions. A late submission is, **by definition**, *any time* past the official deadline, according to the departmental *submission system*.
- Please read this document completely, from start to finish, before beginning work on the assignment.

2 Introduction

The main purpose of this assignment is to add on to your compiler a component of *semantic analysis*. This component shall compute the *lexical addresses* for 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

Please add 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' =
  | Const' of constant
  | Var' of var
  | Box' of var
  | BoxGet' of var
  | BoxSet' of var * expr'
  | If' of expr' * expr' * expr'
  | Seq' of expr' list
  | Set' of expr' * expr'
  | Def' of expr' * expr'
  | Or' of expr' list
  | LambdaSimple' of string list * expr'
  | LambdaOpt' of string list * string * expr'
  | Applic' of expr' * (expr' list)
  | ApplicTP' of expr' * (expr' list);;
```

The type `expr'` will be used by output of the semantic analysis phase of your compiler, 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: first, `annotate_lexical_addresses` will be called, the result of the call will be passed as an argument to the `annotate_tail_calls` function, and finaly, the `expr'`, annotated with tail calls and lexical addresses, will be passed to the `box_set` procedure.

3.2 Lexical addressing

You will provide an implementation for the procedure `annotate_lexical_addresses`, which takes an `expr` and returns an `expr'`, where all `Var` records have been replaced by `Var'` records. If you notice the type constructor `Var'`, you will see that it takes an argument of type `var`, which is a disjoint type made of `VarFree`, `VarParam`, and `VarBound`. So instances of `Var'` should contain 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 insance 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 nither 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 `Applic'` with corresponding instances of `ApplicTP'`. 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

For this assignment, we are going to box any variable that meets the following criteria:

- The variable has [at least] one occurrence in for `read` within some closure,and [at least] one occurrence for `write` in another closure.
- Both occurrences do not already refer to the same rib in a lexical environment

To box a variable `VarParam(v, minor)`, we must do 3 things:

1. Add the expression `Set(VarParam(v, minor), Box'(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.

Notice that there are some cases where the above recipe will box a variable unnecessarily. It is easy to improve our analysis of which variables to box using some extra data-flow analysis, but we shall refrain from doing this for the sake of simplicity.

4 What to submit

In this course, we use the `git` DVCS for assignment publishing and submission. You can find more information on `git` at `https://git-scm.com/`.

To begin your work, pull the updated assignment template from the course website by navigating to your local repository folder from assignment 1 and executing the command: `git pull`

If you haven't work on assignment 1 before starting this work, you can simply clone a fresh copy of the repository by executing the command:

```
git clone https://www.cs.bgu.ac.il/~comp191/compiler
```

This will create a copy of the assignment template folder, named `compiler`, in your local directory. The template contains four (4) files:

- `reader.ml` (assignment 1 interface)
- `tag-parser.ml` (assignment 2 interface)
- `semantic-analyser.ml` (assignment 3 interface)
- `pc.ml` (the *parsing combinators* library)
- `readme.txt`

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.

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 *va'adat mishma'at*, in pursuit of disciplinary action.

Submissions are only allowed through the submission system.

You are required to submit a **patch file** of the changes you made to the assignment template. See instructions on how to create a patch file below.

Please note that any modifications you make to `pc.ml` will be discarded during the testing process, as our testers will use our version of `pc.ml` for the tests.

You are provided with a structure test in order to help you ensure that our tests are able to run on your code properly. Make sure to run this test on your final submission.

4.1 Creating a patch file

Before creating the patch, review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the 'git add' command:

```
git add -Av .; git commit -m "write a commit message"
```

At this point you may review all the changes you made (the patch):

```
git diff origin
```

Once you are ready to create a patch, simply make sure the output is redirected to the patch file:

```
git diff origin > compiler.patch
```

After submission (but before the deadline), it is strongly recommended that you download, apply and test your submitted patch file. Assuming you download `assignment3.patch` to your home directory, this can be done in the following manner:

```
cd ~
git clone https://www.cs.bgu.ac.il/~comp191/compiler fresh_compiler
cd fresh_compiler
git apply ~/compiler.patch
```

Then test the result in the directory `fresh_compiler`.

Finally, remember that your work will be tested on lab computers only! We advise you to test your code on lab computers prior to submission!

4.2 What we will test

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.

5 Tests, testing, correctness, completeness

Please start working on the assignment ASAP! Please keep in mind that:

- We will not (and cannot) provide you with anything close to full coverage. It's your responsibility to test and debug and fix your own code! We're nice, we're helpful, and we'll do our best to provide you with many useful tests, but this is not and cannot be all that you rely on!
- We encourage you to contribute and share tests! Please do not share ocaml code.
- This assignment can be tested using ocaml and Chez Scheme. You don't really need more, beyond a large batch of tests…
- We encourage you to compile your code and test frequently.

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 behaviour presented in class.
3. You did not change the module signatures
4. Your semantic analyser runs correctly under OCaml on the departmental 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 you work in its entirety