

Project 1

CMSC 22620/32620, Spring 2007

Assigned: April 10, 2007

Due: April 17, 2007

1 Introduction

The purpose of the first project is twofold. The main goal is to get a working implementation of the conversion from the `Lambda` language to the ANF (“A-Normal Form”) language. The second goal is to (as a side effect of writing this short phase) familiarize yourself with the two forms of the intermediate language and their semantics.

2 Instructions

2.1 Files

Download the file `project1.tgz` from the course web page. This is a compressed tarball containing the files that define the two intermediate languages (as ML types) and also simple “definitional” interpreters—one for each language—which you can use as a reference whenever you have doubts about the semantics.¹ There is also a file `lambda2anf.sml` which currently contains a skeleton of the code you are supposed to write.

Here is a roadmap for understanding the purpose of each file:

`proj1.cm` This is the CM description file for this project.

`lvar.sml` Defines type `lvar` (which is really just a synonym for `int`). Values of type `lvar` are used to identify the variables used in the two calculi in question (`Lambda` and ANF). The word “lvar” stands for `Lambda VARiable`. Structures `LVar.Set` and `LVar.Map` implement variable sets and finite maps from variables to arbitrary values. Function `new` generates a fresh λ -variable. The string argument is used for pretty-printing purposes. Function `clone` is like `new`, but the string is taken from the name of an existing variable. You do not need to worry about the other functions in this module.

`purity.sml` This module defines a simple “boolean” datatype, i.e., a type with two mnemonically named constructors `Pure` and `Impure`.

¹For no particular reason, the interpreter for `Lambda` is written in direct-style, while the interpreter for ANF is written in continuation-passing style.

`litdata.sml` Type `LitData.integer` is the type that represents the target language's integer values within the compiler. (This is currently `Int32.int`, but we might change this in the future.)

`label.sml` Contains the definition of a type `label` which represents assembly code labels.

`oper.sml` Contains definitions of types for comparison- and arithmetic operators as well as some utility routines for applying these operations to values.

`lambda.sml` Defines the Lambda language consisting of types `value`, `exp`, and `function`.

`anf.sml` Does the same for the ANF language. Lambda and ANF share the same `value` type.

`lambda-interpreter.sml` This is a definitional interpreter for the Lambda language.

`anf-interpeter.sml` Same for the ANF language.

`machspec.sml` Helper file for the two interpreters.

`lambda2anf.sml` The template which you have to fill in.

2.2 SML/NJ

To get started, the first step is to see whether you have access to a working SML/NJ installation. After downloading, uncompressing, and untaring said tarball, you should end up with a directory named `sources` containing the files listed above. Go to that directory and fire up SML/NJ by typing `sml` at the shell prompt. You should see a greeting from SML/NJ and a new input prompt. At this prompt, type `CM.make "proj1.cm"`; . The program should compile. You need to edit `lambda2anf.sml`, replace the dummy body of function `convert` with your solution, and then re-run `CM.make`.

2.3 Testing

To test your code, hand-craft some non-trivial Lambda expression and an accompanying "label environment" (see `lambda-interpreter.sml`)² so that the Lambda interpreter evaluates this expression to, say, an integer value. Then convert the Lambda expression to its ANF equivalent (using the `LambdaToANF.convert` function you implemented). Now send the result to the ANF interpreter (using the same label environment) and confirm that the result is the same as before.

3 Handing it in

In addition to un-commenting the two lines that have been commented out from `proj1.cm`, you should only have to make changes to file `lambda2anf.sml`. To hand in your

²You won't need a non-trivial label environment, so just use, e.g., `fn _ => raise Fail "unbound label"`.

solution, send this file as an e-mail attachment to the instructor using the following e-mail address:

```
instructor | blume (at) tti (hyphen) c (dot) org
```

If you make other changes (which should really not be necessary!), then bundle all your files as a tarball and attach that to your e-mail.

4 Hints

- Use the “meta-continuation passing” technique that was discussed in class.
- Deal with tail-recursion by generating `JUMP` instead of `CALL` wherever you can. As we explained in class, this is probably best done using two separate conversion functions. But you can design your own approach if you prefer.
- Generalize the meta-continuation technique for dealing with lists of expressions. (You will need this twice: once for applications and once for records.)
- Make sure your implementation properly merges control flows for `CMP` rather than causing a potential code explosion during conversion.
- Use `LVar.new` or `LVar.clone` to when you need fresh temporaries during conversion.

5 Extra credit

When inspecting intermediate code (or generated assembly code), some approximate information about names can make it easier to relate such code to the original source code. The `LVar` module can keep name information on a per- λ -variable basis. This information should be tracked all the way to the back end.

To make this work, every translation stage has to try its best to preserve names. Therefore, you should use `LVar.clone` instead of `LVar.new` wherever that makes sense. Design a revised interface for your internal conversion routines so that name information can be passed along efficiently and used conveniently.