CMSC 22620/32620	Implementation of	Project 1
Spring 2015	Computer Languages II	April 16, 2015

Translation to LambdaIR

Due: April 27, 2015

1 Introduction

The second project is to implement a translation from the typed, abstract-syntax tree representation produced by the compiler's front end to a normalized λ-calculus representation that we call `LambdaIR`. You will use the normalization algorithm discussed in class. The datatypes that represent this IR are as follows:

```
datatype exp = Exp of (ProgPt.t * exp_rep)

and exp_rep
  = LETEXP of (var list * exp * exp)
  | FIX of lambda list * exp
  | LET of var * rhs * exp
  | TRY of exp * var * exp
  | IF of var * exp * exp
  | CASE of var * (pat * exp) list * exp option
  | THROW of var
  | APPLY of var * var list
  | RETURN of var list

and rhs
  = PRIM of PrimOp.t * var list
  | CON of LambdaDCon.t * var list
  | ALLOC of var list
  | SELECT of (int * var)
  | CALL of var * var list
  | NUM of IntInf.int
  | STR of string

and lambda = FN of var * var list * exp

and pat = PAT of LambdaDCon.t * var list
```

Note, however, that the the `LambdaIR` module provides *smart constructors*, which you should use to build trees (*e.g.*, use `mkLET(...)` instead of `Exp(ProgPt.new(),LET(...))`).

2 Details

2.1 AST simplification

The seed code contains an AST to AST pass that simplifies the program by converting curried definitions to nested definitions, η expanding where necessary, and expanding applications of the built-in function ignore. This simplification pass (in `proj-2/translate/simplify.sml`) will make it easier to convert the LambdaIR to AST.

For example, data-constructor functions can appear in contexts other than application as in the following code:

```
data Point with con Pt of Integer * Integer;
  ...
    map [Integer * Integer, Point] Pt
      ((1,2)::(3,4)::Nil[Integer*Integer])
  ...
```

In this case, the simplification pass η-converts the occurrence of the data constructor `Pt` and replaces it with the expression

```
{ fun mkPoint (arg : (Integer * Integer)) -> Point = Pt arg; mkPoint }
```

2.2 Data-constructor representation

The `LambdaIR` representation of data constructors includes information about their run-time representation. All **Flang** values are represented by a single machine word. In many cases, this word is a pointer to heap-allocated storage, but it might also be an immediate value. We refer to values that are represented as pointers as *boxed* values, while values that are represented as immediate integers are *unboxed*. As we explain below, the values of a datatype may be both boxed and unboxed, in which case we describe the type as having a *mixed* representation. Since type variables can be instantiated to any type, they have a mixed representation. The builtin `Integer` type maps directly onto unboxed integers and the `String` type is represented as a pointer. Function types are also represented as pointers.

The representation of data constructors is determined as follows. Let T be a data type with n nullary constructors $C_1, \ldots, C_n$ and m data-constructor functions F_1 **of** $\tau_1, \ldots, F_m$ **of** τ_m. The following table gives the representation of the various constructors based on the number of constructors and the representations of the τ's.

n	m	C_i	$F_j(v)$	T's representation
> 0	0	i	n.a.	unboxed
0	1	n.a.	v	τ_1's representation
> 0	1	i	v if τ_j is boxed	mixed
> 0	1	i	$\langle v \rangle$ if τ_j is unboxed or mixed	mixed
0	> 1	n.a.	$\langle j, v \rangle$	boxed
> 0	> 1	i	$\langle j, v \rangle$	mixed

In this chart, i means the immediate (unboxed) value i and $\langle \cdots \rangle$ means a heap-allocated tuple.

Applying this algorithm to the the builtin datatypes we get:

$$\begin{array}{rcl}
\texttt{Unit} & \to & 0 \\
\texttt{False} & \to & 0 \\
\texttt{True} & \to & 1 \\
\texttt{Nil} & \to & 0 \\
a\texttt{::}b & \to & \langle a,\, b\rangle
\end{array}$$

As can be seen from the above table, nullary data-constructors are always represented by unboxed values (called *tags*). Data-constructor functions have three possible representations:

1. *transparent*, where their representation is the same as that of their argument (rows 2 and 3 of the table)
2. *boxed*, where the argument value is contained in a one-word heap object (row 4 or the table)
3. *tagged box*, where the argument value is paired with a tag in a two-word heap object (rows 5 and 6 of the table)

The seed code includes a module, `DataRep`, in `translate/data-rep.sml`, which implements a pass over the AST to compute data-type representations. You should use this module in your translation.

2.3 Pre-defined operators and identifiers

Most of the built-in operators, such as `+`, are translated to *primitive operations* (see the `Prim` structure). The one exception is string concatenation (`@`), which should be mapped to a call to the external function `stringConcat`.

The predefined functions (*e.g.*, `print`) are going to be translated to predefined variables in the `LambdaIR` representation. We will also be translating local AST variables to `LambdaIR` variables. To manage these translations, you will need to pass an environment as an extra argument to your translation functions (see the `TranslateEnv` module).

2.4 Type abstraction and application

The `LambdaIR` representation is not typed, but we must keep the runtime effect of type abstraction in the resulting IR. We do so by using zero-argument functions to represent type abstraction.

In the case of polymorphic data constructors, however, we do not have to preserve type abstraction since they are pure values. For example, consider the following **Flang** code fragment:

```
data Option[a] with
  con None
  con Some of a;
let x : Option[Integer] = None[Integer];
let y : [a] Option[a]   = None;
let z : Option[Boolean] = y[Boolean];
```

Based on the algorithm above, we would represent `None` as the immediate value 0, but the question remains about how to handle the type argument? Since types do not have a run-time significance, we can treat application of a polymorphic data constructor to type arguments as a no-op. This choice means that the right-hand-side of `y`'s binding has to be η-converted as we do for data-constructor functions that are used as values.

```
        let y : [a] Option[a] = {fun mkNone [a] -> Option[a] = None[a]; mkNone};
```

Strictly speaking, the eta-converted `none` function is not legal **Flang** code, since it violates the requirement that functions have at least one value parameter, but we relax this restriction inside the compiler.

2.5 Smart constructors

The `LambdaIR` module defines functions for building `LambdaIR` terms. You should *always* use these functions to construct terms, as they ensure that certain data-structure invariants are preserved.

3 An example

Consider the following LangF program:

```
fun id [a] (x : a) -> a = x;
id [Integer -> Integer] (id [Integer])
```

The AST representation of this program after simplification is

```
LetExp(
  [FunBind(id, [TyParam(-)],
    LetExp(
      [FunBind(id1, [ValParam x], VarExp x)],
      VarExp id1))],
  AppExp(
    TyAppExp(VarExp id, [-]),
    TyAppExp(VarExp id, [-])))
```

It translates into the following LamdaIR term:

```
FIX(
  [FN(id, [], FIX([FN(id1, [x], RETURN[x])], RETURN[id1]))],
  LETEXP([t1], APPLY(id, []),
  LETEXP([t2], APPLY(id, []),
    APPLY(t1, [t2]))))
```

Note also that the type applications are preserved as applications to the empty argument list.

4 Submission

Sources for Project 2 will be seeded into your phoenixforge repositories. These sources include the front-end code, as well as support code for this assignment. You will complete the implementation of the `Translate` module (in `translate/translate.sml`) and commit your solution into the repository. You should not need to change any other file in the source code.

We will collect projects from the SVN repositories at 10pm on Monday, April 27. Please make sure that you have committed your final version before then.

Revision history

April 24, 2015 Fixed typo in η-conversion example.

April 14, 2015 Original version