Assignment 4: Type Extensions

Due: Wednesday, October 25, 2017 at 4:20pm
Submission deadline: Saturday, October 28, 2017 at 4:20pm

Now that you've got some interpeter experience under your belt, it's time to take your language to the next level. In this assignment, you will implement an interpreter for the *exciting simply typed lambda calculus*, or $\lambda_{\rightarrow !}$ (a name of my own invention). Specifically you will extend our previous language $\lambda_{\rightarrow}$ with algebraic data types, pattern matching and polymorphic/existential types.

Setup

Setup is same as always—copy the assignment folder out of our folder on Rice.

On the Rice machines:

```
cp -r /afs/ir/class/cs242/assignments/assign4 assign4
```

On your local machine:

```
scp -r <SUNetID>@rice.stanford.edu:/afs/ir/class/cs242/assignments/assign4 assign4
```

Requirements

You must implement the translation, typechecking, and interpretation stages of the interpreter. You wll turn in these files:

- `translator.ml`
- `typechecker.ml`
- `interpreter.ml`

There are no written portions to this assignment.

Submitting

Then upload the files in your copy of `assign4` to Rice. To submit, navigate to the assignment root and execute:

```
python /afs/ir/class/cs242/bin/submit.py 4
```

Note: We are grading tests that do not result in an error using exact text matching, and will not fix submissions that have extra print statements or the like. It is your responsibility to test before submitting

Language specification

Adding algebraic datatypes, pattern matching, and first-order type logic to our language introduces a substantial level of complexity in both the language and its implementation. A common technique for mitigating complexity in language implementation is to use *intermediate representations* (IRs), or mini-languages internal to the interpreter. For our language $\lambda_{\rightarrow !}$, we have the top-level language defined by what the programmer will write as well an IR for our typechecker and interpreter.

First, we will examine the grammar for our new language. Specifically, we have added new types: `Product`, `Sum`, `ForAll`, `Exists` and the corresponding terms necessary to implement those types. These are precisely the same formulations as we discussed in class. Additionally, this language introduces *pattern matching*, a restricted version of the `match` clause you've used in OCaml.

Type τ ::=	$\texttt{Var}(X)$	X	variable
	$\texttt{Int}$	$\texttt{int}$	integer
	$\texttt{Fn}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$	function
	$\texttt{Product}(\tau_1, \tau_2)$	$\tau_1 * \tau_2$	product
	$\texttt{Sum}(\tau_1, \tau_2)$	$\tau_1 + \tau_2$	sum
	$\texttt{ForAll}(X, \tau)$	$\forall X.\, \tau$	forall
	$\texttt{Exists}(X, \tau)$	$\exists X.\, \tau$	exists

Term t ::=	$\texttt{Var}(x)$	x	variable
	$\texttt{Int}(n)$	n	integer
	$\texttt{Binop}(\oplus, t_1, t_2)$	$t_1 \oplus t_2$	binary operation
	$\texttt{Lam}(x, \tau, t)$	$\texttt{fn}\ (x : \tau)\ .\ t$	function
	$\texttt{App}(t_1, t_2)$	$t_1\ t_2$	application
	$\texttt{Pair}(t_1, t_2)$	(t_1, t_2)	tuple
	$\texttt{Project}(t, d)$	$t.\, d$	projection
	$\texttt{Inject}(t, d, \tau)$	$\texttt{inj}\ t = d\ \texttt{as}\ \tau$	injection
	$\texttt{TLam}(X, t)$	$\texttt{tfn}\ X\ .\ t$	type function
	$\texttt{TApp}(t, \tau)$	$t\ [\tau]$	type application
	$\texttt{TPack}(\tau_1, t, \tau_2)$	$\{\tau_1, t\}\ \texttt{as}\ \tau_2$	type abstraction
	$\texttt{Let}(\rho, t_1, t_2)$	$\texttt{let}\ \rho = t_1\ \texttt{in}\ t_2$	let
	$\texttt{Match}(t, (\rho_1, t_1), (\rho_2, t_2))$	$\texttt{match}\ t\ \{\ \rho_1 \hookrightarrow t_1 \mid \rho_2 \hookrightarrow t_2\ \}$	match

Pattern ρ ::=	$\texttt{Wildcard}$	--	wildcard
	$\texttt{Var}(x, \tau)$	$x : \tau$	variable binding
	$\texttt{Alias}(\rho, x, \tau)$	$\rho\ \texttt{as}\ x : \tau$	alias
	$\texttt{Pair}(\rho_1, \rho_2)$	(ρ_1, ρ_2)	tuple
	$\texttt{TUnpack}(X, x)$	$\{X, x\}$	type unpack

Binop $\oplus$ ::=	$\texttt{Add}$	$+$	addition
	$\texttt{Sub}$	$-$	subtraction
	$\texttt{Mul}$	$*$	multiplication
	$\texttt{Div}$	$/$	division

Direction d ::=	$\texttt{Left}$	L
	$\texttt{Right}$	R

As a refresher from lecture, `Inject` "injects" the term t as the left or right side of the sum type τ, depending on what d is. `TLam` and `TApp` are similar to `Lam` and `App` except that they operate on types, and not terms. That means `TLam` maps a variable name to a type τ and not a term t. `TApp` expects a type τ as an argument for some `TLam`.

`TPack`, `Exists`, and `TUnpack` are intimately related. `TPack` is always used with the type `Exists`. `Exists` states that there exists a type X that looks like the type τ. This essentially allows us to define abstract interfaces. `TPack`, "packs" a concrete implementation of some type τ_1 into some existential type τ_2. `TUnpack` allows us to retrieve the concrete implementation.

As mentioned, all of the rules for algebraic data types and first order type logic mirror the semantics discussed in class, so refer back to those notes for more details. The pattern matching mechanism is novel, however, as this is the first time we will give it a formal treatment. Here's a few examples using patterns in this language:

$$(\texttt{let}\ x : \texttt{int} = 2\ \texttt{in}\ 1 + x) \Longrightarrow 3$$
$$(\texttt{let}\ (a : \texttt{int}, b : \texttt{int})\ \texttt{as}\ c : \texttt{int} * \texttt{int} = (1, 2)\ \texttt{in}\ a + b + c.L) \Longrightarrow 4$$
$$(\texttt{match}\ (\texttt{inj}\ (1, 2) = L\ \texttt{as}\ (\texttt{int} * \texttt{int}) + \texttt{int})\ \{\ (a : \texttt{int}, b : \texttt{int}) \hookrightarrow a + b \mid c : \texttt{int} \hookrightarrow c\ \}) \Longrightarrow 3$$

If we wanted to, we could stop here and define a full statics and semantics for the language above. However, we need not to, since we can *translate* some parts of the language into other. Specifically, we will eliminate pattern matching in a translation pass. For example, we could define the following translation of a tuple match (where $\leadsto$ means "translates to"):

$$\texttt{let}\ (a : \texttt{int}, b : \texttt{int})\ \texttt{as}\ c : (\texttt{int}, \texttt{int}) = (1, 2)\ \texttt{in}\ a + b$$
$$\leadsto$$
$$(\texttt{fn}\ (a : \texttt{int})\ .\ (\texttt{fn}\ (b : \texttt{int})\ .\ (\texttt{fn}\ (c : \texttt{int} * \texttt{int})\ .\ a + b)\ (1, 2))\ (1, 2).\ R)\ (1, 2).\ L$$

To formalize this translation, we will first define a grammar for a new IR:

Type τ ::=	$\texttt{Var}(X)$	X	variable
	$\texttt{Int}$	$\texttt{int}$	integer
	$\texttt{Fn}(\tau_1, \tau_2)$	$\tau_1 \rightarrow \tau_2$	function
	$\texttt{Product}(\tau_1, \tau_2)$	$\tau_1 * \tau_2$	product
	$\texttt{Sum}(\tau_1, \tau_2)$	$\tau_1 + \tau_2$	sum
	$\texttt{ForAll}(X, \tau)$	$\forall X.\ \tau$	forall
	$\texttt{Exists}(X, \tau)$	$\exists X.\ \tau$	exists
Term t ::=	$\texttt{Var}(x)$	x	variable
	$\texttt{Int}(n)$	n	integer
	$\texttt{Binop}(\oplus, t_1, t_2)$	$t_1 \oplus t_2$	binary operation
	$\texttt{Lam}(x, \tau, t)$	$\texttt{fn}\ (x : \tau)\ .\ t$	function
	$\texttt{App}(t_1, t_2)$	$t_1\ t_2$	application
	$\texttt{Pair}(t_1, t_2)$	(t_1, t_2)	tuple
	$\texttt{Project}(t, d)$	$t.\ d$	projection
	$\texttt{Inject}(t, d, \tau)$	$\texttt{inj}\ t = d\ \texttt{as}\ \tau$	injection
	$\texttt{Case}(t, (x_1, t_1), (x_2, t_2))$	$\texttt{case}\ t\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \}$	case
	$\texttt{TLam}(X, t)$	$\texttt{tfn}\ X\ .\ t$	type function
	$\texttt{TApp}(t, \tau)$	$t\ [\tau]$	type application
	$\texttt{TPack}(\tau_1, t, \tau_2)$	$\{\tau_1, t\}\ \texttt{as}\ \tau_2$	type abstraction
	$\texttt{TUnpack}(X, x, t_1, t_2)$	$\texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2$	type open
Binop $\oplus$ ::=	$\texttt{Add}$	$+$	addition
	$\texttt{Sub}$	$-$	subtraction
	$\texttt{Mul}$	$*$	multiplication
	$\texttt{Div}$	$/$	division
Direction d ::=	$\texttt{Left}$	L	
	$\texttt{Right}$	R	

This IR has three differences from the top-level:

1. Patterns are removed entirely.
2. `TUnpack` has moved from `Pattern` to `Term`.
3. `Match` has been replaced by `Case`.

In terms of the code, the first grammar corresponds to the `Lang` module in `ast.ml`, whereas the second corresponds to the `IR` module.

Part 1: Translator

Your first job is to implement the translation function in `translator.ml`:

```
val translate : Lang.Term.t -> IR.Term.t
```

This function takes a term of the top-level language `Lang.Term.t` and converts it into an `IR.Term.t`. It cannot fail, so we do not return a `Result.t`.

The translation is largely trivial for most of the language as the only major difference is the `Lang.Term.Let` and `Lang.Term.Match` cases. We have implemented all of the trivial cases as well the `Match` case for you.

We can formalize the `Let` and `Match` translations by defining a series of translation rules:

$$\frac{t_2 \rightsquigarrow t_2'}{\texttt{let}\ \text{--} = t_1\ \texttt{in}\ t_2 \rightsquigarrow t_2'}\ (\text{Tr-let}_1) \qquad \frac{t_1 \rightsquigarrow t_1' \quad t_2 \rightsquigarrow t_2'}{\texttt{let}\ x : \tau = t_1\ \texttt{in}\ t_2 \rightsquigarrow (\texttt{fn}\ (x : \tau)\ .\ t_2')\ t_1'}\ (\text{Tr-let}_2)$$

$$\frac{t_1 \rightsquigarrow t_1' \quad \texttt{let}\ \rho = t_1\ \texttt{in}\ t_2 \rightsquigarrow t'}{\texttt{let}\ \rho\ \texttt{as}\ x : \tau = t_1\ \texttt{in}\ t_2 \rightsquigarrow (\texttt{fn}\ (x : \tau)\ .\ t')\ t_1'}\ (\text{Tr-let}_3) \qquad \frac{(\texttt{let}\ \rho_1 = t_1 . L\ \texttt{in}\ \texttt{let}\ \rho_2 = t_1 . R\ \texttt{in}\ t_2) \rightsquigarrow t'}{\texttt{let}\ (\rho_1, \rho_2) = t_1\ \texttt{in}\ t_2 \rightsquigarrow t'}\ (\text{Tr-let}_4)$$

$$\frac{t_1 \rightsquigarrow t_1' \quad t_2 \rightsquigarrow t_2'}{\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \rightsquigarrow \texttt{unpack}\ \{X, x\} = t_1'\ \texttt{in}\ t_2'}\ (\text{Tr-let}_5)$$

$$\frac{t \rightsquigarrow t' \quad \texttt{let}\ \rho_1 = x_1\ \texttt{in}\ t_1 \rightsquigarrow t_1' \quad \texttt{let}\ \rho_n = x_2\ \texttt{in}\ t_2 \rightsquigarrow t_2'}{\texttt{match}\ t\ \{\ \rho_1 \hookrightarrow t_1 \mid \rho_2 \hookrightarrow t_2\ \} \rightsquigarrow \texttt{case}\ t'\ \{\ x_1 \hookrightarrow t_1' \mid x_2 \hookrightarrow t_2'\ \}}\ (\text{Tr-match})$$

Look at `Match` for an example of implementing one of these translation rules. After implementing these rules, you will have a complete pipeline to move from source code to your IR.

> Note: if we wanted to formally verify our translation rules, we could define a dynamics for both the toplevel `Lang` and the `IR`, and then prove that for all $t : \tau$ that if $t_L \Rightarrow t_L'$ in `Lang` and $t_L \rightsquigarrow t_{IR}$ then $t_{IR} \Rightarrow t_{IR}'$ implies that $t_L' \rightsquigarrow t_{IR}'$.

Part 2: Typechecker

Your next task is to implement a typechecker for the IR. Because we decided to translate before typechecking, this will reduce the complexity of the typechecker (although this will obscure type errors–it's a tricky business to match a type error on generated code back to the original source).

The format of the typechecker is the mostly same as the previous assignment, but now the type system is more complex. The statics are:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{T-var}) \qquad \frac{}{\texttt{Int}(n) : \texttt{int}}\ (\text{T-n}) \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\texttt{fn}\ (x : \tau_1)\ .\ t) : \tau_1 \to \tau_2}\ (\text{T-fn})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}\ (\text{T-app}) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2}\ (\text{T-tuple})$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t . L : \tau_1}\ (\text{T-project-L}) \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t . R : \tau_2}\ (\text{T-project-R})$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \texttt{inj}\ t = L\ \texttt{as}\ \tau_1 + \tau_2 : \tau_1 + \tau_2}\ (\text{T-inject-L}) \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \texttt{inj}\ t = R\ \texttt{as}\ \tau_1 + \tau_2 : \tau_1 + \tau_2}\ (\text{T-inject-R})$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \texttt{case}\ t\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \} : \tau}\ (\text{T-case})$$

$$\frac{\Gamma, X \vdash t : \tau}{\Gamma \vdash \texttt{tfn}\ X\ .\ t : \forall X. \tau}\ (\text{T-tfn}) \qquad \frac{\Gamma \vdash t : \forall X. \tau_1}{\Gamma \vdash t\ [\tau_2] : [X \to \tau_2]\ \tau_1}\ (\text{T-tapp}) \qquad \frac{\Gamma \vdash t : [X \to \tau_1]\ \tau_2}{\Gamma \vdash \{\tau_1, t\}\ \texttt{as}\ \exists X. \tau_2 : \exists X. \tau_2}\ (\text{T-pack})$$

$$\frac{\Gamma \vdash t_1 : \exists X. \tau_1 \quad \Gamma, X, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : \tau_2}\ (\text{T-unpack})$$

There are two major additions: sum/product types and polymorphic/existential types. Typechecking the former is straightforwardly derived from the rules like from the previous assignment, but the latter introduces a new concept: type variables. Now, your typing context keeps track of two things:

1. Mappings from term variables to types, notated by $\Gamma, x : \tau \vdash e : \tau$. This is as before used for when term variables are introduced by lambdas, or here also case expressions.
2. Existence of type variables, notated by $\Gamma, X \vdash e : \tau$. Type variables do not map to anything, the context just tracks when a type variable is in scope (to catch type expressions that use an undefined type variable).

Concretely, you will implement the following function:

```
val typecheck_term : String.Set.t -> Type.t String.Map.t -> Term.t -> Type.t
```

Where `typecheck_term type_env env t` returns the type of `t` if it exists. Note that instead of using a `Result.t`, this time you can use a simpler approach: whenever a type error occurs, you should raises a `TypeError` exception, e.g. `raise (TypError "Does not typecheck")`, with the appropriate error message. Otherwise, assume that when you call `typecheck_term` recursively you get back a valid type.

Here, `env` is the same mapping from variables to types as in the previous assignment, but we have added `tenv` which is a set of type variables. Type variables do not map to anything (they do not themselves have types), but instead just provide the context that a type variable is in scope during typechecking of an expression.

We have provided you a function `typecheck_type : String.Set.t -> Type.t -> Type.t`. Whenever you're typechecking a term that contains a type specified by the user, for example the type annotation on a function or an injection, you will need to check that the provided type does not use unbound type variables. For example, the following expression should not typecheck because Y is unbound:

$$\texttt{tfn}\, X\, .\ \texttt{fn}\, (x : Y)\, .\ x$$

Lastly, during typechecking, there will be points where you will want to compare the equality of two types. *Do not* use the equals operator (e.g. `tau1 = tau2`) as that would be incorrect! Use the `Type.aequiv` function that compares two types for equality *up to renaming of bound variables*, or check them for *alpha-equivalence*. For example, the types $\forall X.\, X$ and $\forall Y.\, Y$ should be considered equal, however the `=` operator will return false because the variable names are different. `Type.aequiv` has the signature `Type.t -> Type.t -> bool`.

> Note: Technically, it's possible for `match` statements to introduce variables that are already free variables in the expression, which is not legal. However, for the sake of this assignment, assume that `match` statements will always use fresh variables. That is, variables introduced in cases will not be found elsewhere in the expression.

Part 3: Interpeter

Lastly, you must implement the interpreter. At this point, all of the terms relating to type abstraction (polymorphic/existential types) have trivial dynamics, since we've already verified our required type properties in the typechecker. Most of the legwork is in supporting algebraic data types. The dynamics are as follows:

$$\frac{}{\texttt{Int}(n)\ \textsf{val}}\ (\text{V-n}) \qquad \frac{}{(\texttt{fn}\ (x:\tau_1)\ .\ e)\ \textsf{val}}\ (\text{V-fn}) \qquad \frac{t_1 \mapsto t_1'}{(t_1\ t_2) \mapsto (t_1'\ t_2)}\ (\text{D-app}_1) \qquad \frac{t_1\ \textsf{val} \quad t_2 \mapsto t_2'}{(t_1\ t_2) \mapsto (t_1\ t_2')}\ (\text{D-app}_2)$$

$$\frac{t_2\ \textsf{val}}{((\texttt{fn}\ (x:\tau)\ .\ t_1)\ t_2) \mapsto [x \to t_2]\ t_1}\ (\text{D-app}_3) \qquad \frac{t_1 \mapsto t_1'}{t_1 \oplus t_2 \mapsto t_1' \oplus t_2}\ (\text{D-binop}_1) \qquad \frac{t_1\ \textsf{val} \quad t_2 \mapsto t_2'}{t_1 \oplus t_2 \mapsto t_1 \oplus t_2'}\ (\text{D-binop}_2)$$

$$\frac{}{t_1\ /\ \texttt{Int}(0)\ \textsf{error}}\ (\text{D-div}) \qquad \frac{}{\texttt{Int}(n_1) \oplus \texttt{Int}(n_2) \mapsto \texttt{Int}(n_1 \oplus n_2)}\ (\text{D-binop}_3)$$

$$\frac{t_1 \mapsto t_1'}{(t_1, t_2) \mapsto (t_1', t_2)}\ (\text{D-tuple}_1) \qquad \frac{t_1\ \textsf{val} \quad t_2 \mapsto t_2'}{(t_1, t_2) \mapsto (t_1, t_2')}\ (\text{D-tuple}_2) \qquad \frac{t_1\ \textsf{val} \quad t_2\ \textsf{val}}{(t_1, t_2)\ \textsf{val}}\ (\text{D-tuple}_3)$$

$$\frac{t \mapsto t'}{t.\,d \mapsto t'.\,d}\ (\text{D-project}_1) \qquad \frac{(t_1, t_2)\ \textsf{val}}{(t_1, t_2).\,L \mapsto t_1}\ (\text{D-project}_2) \qquad \frac{(t_1, t_2)\ \textsf{val}}{(t_1, t_2).\,R \mapsto t_2}\ (\text{D-project}_3)$$

$$\frac{t \mapsto t'}{\texttt{inj}\ t = d\ \textsf{as}\ \tau \mapsto \texttt{inj}\ t' = d\ \textsf{as}\ \tau}\ (\text{D-inject}_1) \qquad \frac{t\ \textsf{val}}{\texttt{inj}\ t = d\ \textsf{as}\ \tau\ \textsf{val}}\ (\text{D-inject}_2)$$

$$\frac{t \mapsto t'}{\texttt{case}\ t\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \} \mapsto \texttt{case}\ t'\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \}}\ (\text{D-case}_1)$$

$$\frac{t\ \textsf{val}}{\texttt{case inj}\ t = L\ \textsf{as}\ \tau\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \} \mapsto [x_1 \to t]\ t_1}\ (\text{D-case}_2)$$

$$\frac{t\ \textsf{val}}{\texttt{case inj}\ t = R\ \textsf{as}\ \tau\ \{\ x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\ \} \mapsto [x_2 \to t]\ t_2}\ (\text{D-case}_3)$$

$$\frac{}{\texttt{tfn}\ X\ .\ t \mapsto t}\ (\text{D-tfn}) \qquad \frac{}{t\ [\tau] \mapsto t}\ (\text{D-tapp}) \qquad \frac{}{\{\tau_1, t\}\ \textsf{as}\ \tau_2 \mapsto t}\ (\text{D-pack})$$

$$\frac{}{\texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \mapsto [x \to t_1]\ t_2}\ (\text{D-unpack})$$

As in the previous assignment, you will implement these dynamics in the `trystep` function in `interpreter.ml`:

```
val trystep : Term.t -> outcome
```

The dynamics for `Int` and `Var` as well as the dynamics for the polymorphic/existential cases are done for you.

Testing

To build the interpreter, run `make`. This will create an executable, `main.byte`. This creates OCaml bytecode that needs to passed to `ocamlrun` to execute. We've provided two scripts `user.sh` and `solution.sh` for `main.byte` and `solution.byte` respectively.

Running `run_tests.py` will run the test suite over all the files in the `tests` directory and compare them to our solution.

```
python3 run_tests.py
```

Note that the tests are just a representative sample of the tests we will run your code on.

To test an individual file, you can use `user.sh` to invoke the interpreter manually, e.g.

```
./user.sh tests/exists1.lam2
```

For each of the functions you have to implement, we have provided a few additional unit tests inside of an `inline_tests` function in each file. Uncomment the line that says:

```
let () = inline_tests ()
```

And then execute `./user.sh` to run those tests. If it does not fail with an assertion error, you have passed those tests.