Lab 3
Negative Types

15-417/817: HOT Compilation
Frank Pfenning

Due Fri Feb 22 (tests), Thu Feb 27 (compilers)
150 points

In this third lab we study the first version of our source language ND with a first-class functions and objects. These extensions also have to be reflected into Sax. The extensions will continue to be the core of our compiler throughout the semester. In brief, you will implemented functions and lazy records, but not yet closure conversion. Your new compilers should produce `.sax` files that are then executed by our reference implementation.

1 Submissions

Your submissions should be handed in directly to Gradescope from Github or Bitbucket. You may hand in as often as you like.

1.1 Test Cases (30 points)

Your handin should have a directory `tests/` that contains 10 distinct test files with a variety of ND programs `<file>.nd`. Your files should have a mix of negative tests (which are required to fail) and positive tests. Among the positive tests should be definitions with parameters and those without. We will continue to use the script `~fp/bin/nd-split` to split each file into several files `<file>_<NN>.nd` containing programs that no longer contain `fail` definitions (as described below); some of these programs may pass all static checks and some not.

Your compilers will parse and perform static checks (including typechecking) on each file resulting from splitting. Those that pass should then be compiled to `<file>_<NN>.nd.sax`. As in Lab 2, parameterless definitions in the target file will be executed by the reference implementation, and the resulting `<file>_<NN>.nd.sax.val` compared to the reference values. You may validate your test files using the reference implementation of ND available at `~fp/bin/nd` and `~fp/bin/nd-test` on the linux.andrew machined.[1]

1.2 Compiler (120 points)

Your handin should contain a `Makefile` at the top level that compiles your sources when invoking `make nd` to create the executable `./nd`.

This ND executable should take a single `<file>.nd` as an argument and write a file `<file>.nd.sax` if static checking succeeds. It may be empty if there are no definitions in `<file>.nd`.

[1] Availability will be announced on Ed Discussion.

2 Grammars

2.1 Lexical Analysis

Except for a few additional keywords, this is the same as for Lab 2.

```
% ... \n or % ... <eof> for single-line comments
(* ... *) for multi-line (nested) comments

<whitespace> ::= [ \t\r\n]

<idstart> ::= [a-zA-Z_]
<idchar> ::= [a-zA-Z_0-9]
<id> ::= <idstart> <idchar>*
<label> ::= ' <idchar>+

<keywords> ::= 'type' | 'defn' | 'fail'
             | 'match' | 'with' | 'end'
             | 'proc' | 'read' | 'write' | 'cut' | 'id' | 'call' | 'reuse'
             | 'value'
             | 'fun' | 'record'     % new in Lab 3
```

Keywords cannot be used as identifiers `<id>`. The character `$` is legal in identifiers in Sax but **not** in ND. This allows you to generate fresh names without fear of conflicting with the source. Similarly, we have declared the keywords of Sax as keywords to avoid unpleasant needs to rename variables in the translation.

2.2 ND Grammar (files *.nd)

```
<testfile> ::= <test>*

<test> ::= <defn>
         | 'fail' <defn>

<prog> ::= <defn>*

<defn> ::= 'type' <id> '=' <tp>
        |  'defn' <id> <parm>* ':' <tp> '=' <exp>

<exp> ::= <id>                      % change for Lab 3
        | <id> <atom>+
        | '(' <exp> ')'             % change for Lab 3
        | '(' ')'                   % change for Lab 3
        | <exp> ',' <exp>
        | <label> <exp>
        | 'match' <exp> 'with' <branch>+ 'end'
        | 'fun' <id> '=>' e         % new in Lab 3
        | 'record' <field>+ 'end'   % new in Lab 3
```

```
<atom> ::= <id>
         | '.' <label>              % new in Lab 3
         | '(' ')'
         | '(' <exp> ')'

<branch> ::= '|' <pat> '=>' <exp>

<field> ::= '|' <label> '=>' <exp> % new in Lab 3

<pat> ::= <id>
        | <pat> , <pat>
        | '(' ')'
        | <label> <pat>
        | '(' <pat> ')'

<parm> ::= '(' <id> ':' <tp> ')'

<tp> ::= '+' '{' <alts> '}'
       |  <tp> '*' <tp>
       | '1'
       | <id>
       | '(' <tp> ')'
       | '&' '{' <alts> '}'        % new in Lab 3
       | <tp> '->' <tp>            % new in Lab 3

<alts> ::= <alt>
        |  <alt> ',' <alts>

<alt> ::= <label> ':' <tp>
```

- `'*'` and `'->'` are right associative, where `'*'` has higher precedence than `'->'` so
  `A * B * C -> D -> E == (A * (B * C)) -> (D -> E)`
- `','` is right associative, so `x, y, z == x, (y, z)`
- `'=>'` has higher precedence than `','`, so
  `(fun x => x, fun y => y) == ((fun x => x), (fun y => y))`
- `<label>` is a prefix with higher priority than `','` and `'=>'`, so
  `'cons x, y == ('cons(x), y)` and
  `'succ 'zero () == 'succ ('zero ())`
- The keyword `'fail'` appears only in the test case sources and **never** in programs seen by your compilers. For a description of splitting, see the Lab 2 spec.

2.3 Statics

The static checks before type-checking remain the same as far Lab 2.

2.4 Typing

The core of the typing rules can be found in Lecture 7.

There are some subtleties regarding type-checking of expressions parsed as `<id> <atom1> ... <atomn>`. How you resolve these depends on your internal data type of expressions (which you may of course modify from the starter code). Technically, the sequence of atoms $a_1 \dots a_n$ form a *spine* [Cervesato and Pfenning, 2003], and it is certainly possible to organize the statics, dynamics, and compilation around them. Or you can translate them to the more common, left-nested form:

- $F\ e_1 \dots e_n$ becomes $F\ [e_1, \dots, e_n]$ when F is a top-level function (that is, a metavariable) and the arguments must all be expressions.
- $x\ a_1 \dots a_n$ becomes $(((x\ a_1)\ a_2) \dots a_n)$ where each a_i is either an expression (so that the juxtaposition becomes function application) or a projection $.k$ (so that the juxtaposition becomes a projection from a record).

The subtleties regarding nested pattern matching do not change from Lab 2 because we can only match a value of negative type with a variable.

The coinductive rules for subtyping $A \leq B$ are given in Lecture 7. These extend those from Lab 2.

2.5 Dynamics

We recommend, but do not require, that you implement a direct evaluator for ND, which you may use to test your code.

2.6 Optimizations

You may consider any hold-over optimization from Lab 2, but we do not require any. If so, the cut/id optimizations seem to be simplest and most immediately valuable.

3 Changes to Sax

One of the biggest changes is that we no longer support the `'fail'` keyword. To test Sax directly, we imagine splitting the source as for ND.[2] This greatly simplifies the static requirements enumerated in Lab 1. We also return to full α-conversion, allowing shadowing not only among sources but also the destination. These are changes you can make, but they will not be tested by the autograder since we are running the code you produce through our own implementation of Sax for this Lab.

The change regarding closures below is postponed to a future lab.

Closures, through closure conversion, have been lifted to the top level, where we have a new kind of definition:

$$\textbf{clos}\ G\ (d : A)\ (y_1 : B_1) \dots (y_n : B_n) = \textbf{write}\ d\ K$$

where

$$\begin{array}{rrcll} \text{Continuations} & K & ::= & (x, y) \Rightarrow P & (A \to B) \\ & & \mid & \{\ell(x_\ell) \Rightarrow P_\ell\}_{\ell \in L} & (\&\{\ell : A_\ell\}_{\ell \in L}) \end{array}$$

[2] We will post on Ed Discussion when such a script becomes available.

These should be the only writing occurrence for continuations. G will close the continuation K over the environment $(y_1, \ldots, y_n)$ and write the resulting pair $\langle (y_1, \ldots, y_n), K \rangle$ to destination d. At runtime, $y_1, \ldots, y_n$ will be addresses.

Types and subtyping are shared between ND and Sax, so the extensions to the type system apply to both.

3.1 Sax Grammar (files `*.sax`)

Unfortunately, the grammar is not entirely backward compatible because the command

```
'read' <id> <pat>
 <cmd>
```

would introduce some ambiguity. It now has to be written in its less compact form

```
'read' <id> '{'
'|' <pat> '=>' <cmd>
'}'
```

Fortunately, you should rarely (if at all) write Sax programs by hand any more.

```
<prog> ::= <defn>*

<defn> ::= 'type' <id> '=' <tp>
         |  'proc' <id> <parm> <parm>* '=' <cmd>

<cmd> ::= 'read' <id> <storable>            % change for Lab 3
        |  'write' <id> <storable>          % change for Lab 3
        |  'cut' <id> ':' <tp> <cmd> <cmd>
        |  'reuse' <id> '=' <id> ':' <tp> <cmd> <cmd>
        |  'id' <id> <id>
        |  'call' <id> <id> <id>*
        |  '{' <cmd> '}'

<storable> ::= <pat>                        % Lab 3
             |  '{' <branch>+ '}'           % Lab 3

<branch> ::= '|' <pat> '=>' <cmd>

<pat> ::= <label> '(' <id> ')'
        |  '(' <id> ',' <id> ')'
        |  '(' ')'

<parm> ::= '(' <id> ':' <tp> ')'
```

References

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13 (5):639–688, 2003.