Lab 4
Adjoint Types

15-417/817: HOT Compilation
Frank Pfenning

Due Thu Mar 20 (tests), Thu Mar 27 (compilers)
150 points

In this fourth lab we study the adjoint version of our source language ND and the intermediate language Sax. Your compilers will 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 Labs 2 and 3, 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 3.

```
% ... \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'
             | 'inst'              % new in Lab 4
             | 'susp' | 'force'    % new in Lab 4
```

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> '[' <mode>+ ']' '=' <tp>  % changed for Lab 4
        |  'defn' <id> <parm>* ':' <tp> '=' <exp>
        |  'inst' <id> <parm>* ':' <tp>  % new in Lab 4

<exp> ::= <id>
        | <id> <atom>+
        | '(' <exp> ')'
        | '(' ')'
        | <exp> ',' <exp>
        | <label> <exp>
        | 'match' <exp> 'with' <branch>+ 'end'
```

```
        | 'fun' <id> '=>' e
        | 'record' <field>+ 'end'
        | '<' <exp> '>'              % new in Lab 4
        | 'susp' e                   % new in Lab 4

<atom> ::= <id>
         | '.' <label>
         | '(' ')'
         | '(' <exp> ')'
         | '.' 'force'               % new in Lab 4
         | '<' <exp> '>'             % new in Lab 4

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

<field> ::= '|' <label> '=>' <exp>

<pat> ::= <id>
        | <pat> , <pat>
        | '(' ')'
        | <label> <pat>
        | '(' <pat> ')'
        | '<' <pat> '>'              % new in Lab 4

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

<tp> ::= '+' '{' <alts> '}'
      |  <tp> '*' <tp>
      | '1'
      | <id> [ '[' <mode>+ ']' ] % changed for Lab 4
      | '(' <tp> ')'
      | '&' '{' <alts> '}'
      | <tp> '->' <tp>
      | '<' <tp> '>'               % new in Lab 4
      | '^' <tp>                   % new in Lab 4
      | '[' <mode> ']' <tp>        % new in Lab 4

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

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

<mode> ::= <id>                    % new in Lab 4
```

Operator Precedence

(Lab 3+) `'*'` and `'->'` are right associative, where `'*'` has higher precedence than `'->'` so

```
A * B * C -> D -> E == (A * (B * C)) -> (D -> E)
```

(Lab 4) `'^'` and `'[' <mode> ']'` are prefix operators with higher precedence than `'*'` and `'->'`

(Lab 2+) `','` is right associative, so `x, y, z == x, (y, z)`

(Lab 3+) `'=>'` has higher precedence than `','` so
`(fun x => x, fun y => y) == ((fun x => x), (fun y => y))`

(Lab 2+) `<label>` is a prefix with higher priority than `','`, and `'=>'`, so
`'cons x, y == ('cons(x), y)` and
`'succ 'zero () == 'succ ('zero ())`

(Lab 4) The prefix operator `'susp'` has the same precedence as `<label>`.

(Lab 2+) 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 and Mode/Type Checking

In addition to the static checks from Lab 2, we have the following requirements.

1. Mode variables are given as lowercase identifiers (starting with character `a` through `z`).
2. Mode constants are given as uppercase identifiers (starting with character `A` through `Z` or _). In this lab, the only valid mode constants are U, A, S, and L (see below).
3. In type definitions

   ```
   type t[m k1...kn] = A
   ```

   we say that m is the mode of t, and the k_i are additional mode parameters. We require all of the following.

   (a) All of m, k_i must be *distinct mode variables*. All mode variables in A must be among m and k_i.

   (b) No mode constants appear in A.

   (c) There must be unique modes for all upshift and downshift modalities in A.

   (d) A type expression `[m]A` requires A to have mode m. This just (potentially) disambiguates and does not otherwise affect the structure of A.

   (e) The only constant modes are U, A, S, and L where $\mathsf{U} > \mathsf{A}$, $\mathsf{U} > \mathsf{S}$, $\mathsf{U} > \mathsf{L}$, $\mathsf{A} > \mathsf{L}$, and $\mathsf{S} > \mathsf{L}$ with $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$, $\sigma(\mathsf{A}) = \{\mathsf{W}\}$, $\sigma(\mathsf{S}) = \{\mathsf{C}\}$, and $\sigma(\mathsf{L}) = \{\,\}$.

   (f) Any instantiation of `t[M K1...Kn]` with mode constants M and K_i in the program must satisfy the presupposition on every shift in A that the upper index is greater or equal to the lower index.

   (g) Occurrences of t in definitions or instantiations without explicit mode arguments become `t[U U...U]`, that is, U functions as the *default mode*.

   (h) Similarly, in definitions or instantiations ambiguous modes are resolved to the default mode U. This does not apply to type definitions (which are in any case not allowed to contain mode constants).

4. Next consider definitions

```
defn F (x1 : A1) ... (xm : Am) : A = e
```

(a) New: All the types A and A_i must contain only mode constants (not mode variables).[2]

(b) A metavariable F has at most one definition, but may have multiple instantiations.

(c) New: A definition implicitly induces an instantiation

```
inst F (x1 : A1) ... (xm : Am) : A
```

5. Next consider instantiations

```
inst F (x1 : B1) ... (xm : Bm) : B
```

(a) All the types B_i and B should only use mode constants.

(b) Each instance declaration requires the definition e of F to be type-checked again with the specific given types (including the specific given modes).

(c) Each call to a metavariable G in e may have one of the given type instantiations.

(d) New: A synthesizing expression e may have more than one possible type B such that $e \Longrightarrow B$. Instead of backtracking over these choices, you should choose the lexically first instance of G such that e may synthesize B with $B \leq A$ when checking $e \Longleftarrow A$, even if the arguments of G may not check against this instances argument types.

(e) Each instance declaration generates a separate Sax procedure `F$j` for the jth instance declaration for F (starting at 0).

Optional for this lab:[3]

- Each call to G in e must be disambiguated to call the correct instance of G. When disambiguating $G\ e_1\ \ldots\ e_n \Longleftarrow A$ we select the *lexically first* instance of G of the form above such that $B \leq A$, as explained for the typechecker.

2.4 Typing

The core of the typing rules can be found in Lecture 10, augmented with the coinductive rules for subtyping $A \leq B$ from Lecture 7. The latter are extended covariantly to the shift modalities.

The subtleties regarding nested pattern matching do not change from Lab 2, except that we have the new pattern $\langle p \rangle$ with the straightforward rules to filter the matching branches.

New in the revised spec: Unlike Lab 3, the expression ( ) does not synthesize a type because its mode would be ambiguous.

3 Changes to Sax

Types and subtyping are shared between ND and Sax, so the extensions to the language of types apply to both.

Adoint typechecking rules for Sax can be found in Lecture 14[4]. Each instance declaration from ND becomes a separate procedure definition in Sax, so there is no overloading of definitions or possible ambiguities regarding modes.

[2] *This is a change from an earlier space that required these to be default modes.*

[3] *This was part of the original spec.*

[4] Availability will be announced on Ed Discussion.

3.1 Sax Grammar (files `*.sax`)

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

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

<cmd> ::= 'read' <id> <storable>
        |  'write' <id> <storable>
        |  'cut' <id> ':' <tp> <cmd> <cmd>
        |  'reuse' <id> '=' <id> ':' <tp> <cmd> <cmd>
        |  'id' <id> <id>
        |  'call' <id> <id> <id>*
        |  '{' <cmd> '}'

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

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

<pat> ::= <label> '(' <id> ')'
        |  '(' <id> ',' <id> ')'
        |  '(' ')'
        |  '<' <id> '>'            % new for Lab 4

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