Lab 2
Natural Deduction and Compilation

15-417/817: HOT Compilation
Frank Pfenning

Due Thu Feb 6 (tests), Thu Feb 13 (compilers)
150 points

In this second lab we study the first version of our source language ND with a linear type system. We further restrict ourselves to the first-order fragment where all types are positive, augmented with metavariables that internalize the typing judgment. Extensions will continue to be the core of our compiler throughout the semester. Finally, you will have a chance to implement the first optimizations!

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 with without. We will provide a script `~fp/bin/nd-split` to split each file into several files `<file>.<i>.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>.<i>.nd.sax`. As in Lab 1, parameterless definitions in the target file will be executed by the reference implementation, and the resulting `<file>.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]

Some of your test files should implement some interesting algorithms. Efficiency of execution is becoming somewhat of an issue, so some passing test cases should perform nontrivial computation.

[1] Availability will be announced on Ed Discussion.

1.2 Compiler (120 points)

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

There is a parser available in OCaml and Rust. You may use another implementation language, but you should contact the course staff to make sure it is available in the correct configuration in the autograder on Gradescope.

2 Grammars

2.1 Lexical Analysis

We change the syntax for comments from Sax to be more typical for functional languages.

```
% ... \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'
```

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> ::= <atom>
```

```
        | <id> <atom>+
        | <exp> ',' <exp>
        | <label> <exp>
        | 'match' <exp> 'with' <branch>+ 'end'

<atom> ::= <id>
         | '(' ')'
         | '(' <exp> ')'

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

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

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

<tp> ::= '+' '{' <alts> '}'
      |  <tp> '*' <tp>
      | '1'
      | <id>
      | '(' <tp> ')'

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

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

`'*'` is right associative, so `A * B * C == A * (B * C)`
`','` is right associative, so `x, y, z == x, (y, z)`
`<label>` is a prefix with higher priority than `','`, 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. They are used to split up the source into several separate files. A *segment* is a sequence `<defn>* 'fail' <defn>` which is written to a separate file containing `<defn>* <defn>`. We then eliminate `'fail' <defn>` and continue to process the same file. For example,

```
 1 fail
 2 type nat = +{zero : 1, 'succ : nat}
 3 type nat = +{'zero : 1, 'succ : nat}
 4
 5 defn succ (x : nat) : nat = 'succ x
 6 fail
 7 defn pred (x : nat) : nat = match x with
 8 | 'succ(x) => x
 9 end
```

will create three files:

1. With just line 2 (should fail)
2. With lines 3, 4, 5, 7, 8, 9 (should fail)
3. With lines 3, 4, 5 (should succeed)

Here we have put `fail` on separate lines to more easily describe the outcome of splitting. You may also just submit files without any `fail` declarations.

2.3 Statics

As in Sax, all types and definitions may be mutually recursive, respectively. Bound variables can be renamed arbitrarily, except those bound "simultaneously" (that is, parameters to a top-level function or variables in a pattern) must be distinct. In addition to linear typing as detailed in the lecture notes, we have the following static requirements.

1. Sums must be nonempty (enforced by the grammar)
2. Sums may not contain any duplicate labels
3. Type definitions must be contractive, that is, their right-hand side cannot be a type name
4. Branches must be nonempty empty (enforced by the grammar)
5. Type names may be defined at most once
6. Type names that are used must be defined
7. Variables bound in a pattern must all be distinct
8. Top-level functions may be defined at most once
9. Top-level functions that are called must be defined
10. All parameters to a top-level function must be distinct

2.4 Typing

The core of the typing rules can be found in Lecture 3. In these rules, the order of antecedents is seen as irrelevant, and the comma operator conjoins contexts with disjoints sets of variables. Shadowing is allowed which could be implemented via renaming of bound variables, or via keeping the contexts ordered in your implementation. In particular, top-level function names and variable names occupy the same name space, and bound variables may shadow metavariables. This is because otherwise occurrence of metavariables without arguments might be ambiguous.

There are some subtleties regarding the type-checking of nested pattern matching, explained in Lectures 5 and 6. In addition, we decided against a separate syntax for sequences of patterns, reusing the syntax for pairs instead (that is $e_1, \ldots, e_n$ for the subjects of a match and $p_1, \ldots, p_n$ for the patterns). This means that we have additional rules for bidirectional typing, where

$$\frac{\Gamma \vdash e_1 \Longrightarrow A \quad \Delta \vdash e_2 \Longrightarrow B}{\Gamma\,;\,\Delta \vdash (e_1, e_2) \Longrightarrow A \otimes B}\otimes I^{\Rightarrow} \qquad \frac{}{\Gamma\,;\,\Delta \vdash (\,) \Longrightarrow \mathbf{1}}\,\mathbf{1}I^{\Rightarrow}$$

We will have occasion to revise this later on. We recommend you use these only to type the subjects of matches, and otherwise use the checking judgment for pairs and unit.

The coinductive rules for subtyping $A \leq B$ are given in Section 5 of Lecture 2. These are the same as for Lab 1, but are applied only in the rule $\Rightarrow/\Leftarrow$.

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

We recommend, but do not require at this point, that you implement the cut/id and reuse optimizations as a form of Sax to Sax transformation. While executing your Sax code, we will measure various dynamic aspects like the number of allocations, size of the heap, steps executed, etc. These will be written as comments in the `<file>.nd.sax.val` files that our interpreter produces.

For this purpose, the folder with the implementation should contain a `readme.txt` or `readme.md` file that explains which optimizations you implemented and any other significant information about your implementation.