OPLSS 2017 PL Background Day 1

Robert Harper and Dan Licata

These exercises are based on materials for Carnegie Mellon 15-312: Principles of Programming Languages; we thank many intructors and teaching assistants for contributing to the design and implementation of these problems.

Here are some problems you can use to review the first day's materials. Please note that we definitely do *not* expect you to finish all of these before tomorrow! Students come to OPLSS with very different backgrounds, and our only goal is that *everyone* learns *something*. If you haven't seen much of this material before, you might spend several days's hands-on sessions working on this assignment. If you've seen the basics before, there are more advanced problems later in this assignment, and more research-level courses coming up later in the week.

The online preview of PFPL is here:

`http://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf`

Paper and pencil

1. Problem 1: This problem asks you to write some programs in Gödel's T.
2. Problem 2: This problem asks you to prove determinism and the correspondence between an evaluation relation and a structural operational semantics. If you have never done proofs by rule induction before, start here.
3. Problem 3: This problem asks you to do some type safety (progress and preservation) proofs and to learn about product and sum types. If you haven't done many proofs by rule induction, or haven't seen progress and preservation before, you should start here, before doing the logical relations proofs.
4. Problem 4: This problem asks you to practice some of the materials from the logical relations lecture.

Implementation Problem 5: This problem asks you to implement a type checker and operational semantics for a simple functional programming language with labeled sums and products (see PFPL Chapters 10 and 11 specifically). If you haven't seen type systems and operational semantics before, implementing them is a good way to really understand them. If you've written a type checker or an interpreter for a functional language, your time might be better spent on the logical relations proofs.

1 Programming in Gödel's T

For this section, work in Gödel's T (see PFPL Chapter 9).

Task 1.1. Define an addition function $\texttt{plus} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$.

Task 1.2. Define a multiplication function $\texttt{mult} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$.

Task 1.3. Define a factorial function $\texttt{fact} : \texttt{nat} \to \texttt{nat}$.

Task 1.4. It is possible to define Gödel's T with either *iteration* or *recursion*. The difference is

$$\frac{\Gamma \vdash e : \texttt{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}(e)\{\texttt{z} \hookrightarrow e_z \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\} : \tau} \qquad \frac{\Gamma \vdash e : \texttt{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{iter}(e)\{\texttt{z} \hookrightarrow e_z \mid y \hookrightarrow e_1\} : \tau}$$

That is, in "iteration", the successor case does not bind a variable standing for the predecessor of the number, only for the recursive call, while in "recursion," it does bind a variable standing for the predecessor. Iteration is clearly a special case of recursion, where the variable goes unused.

Perhaps surprisingly, it is also possible to show that conversely, recursion can be defined from iteration. Show how to code recursion using iteration in Gödel's T *with product types* (do the next problem first if you don't know how those work): define the recursor and show that the instruction steps for executing the recursor on $\texttt{z}$ and $\texttt{s}(n)$ are simulated by one or more instruction steps in the language with iteration.

2 Rule Induction Warm-Up

For this problem, again consider Gödel's T as described in PFPL Chapter 9, i.e. a language with function types $\to$ and natural numbers.

Determinism of Stepping **Task 2.1**. Prove that stepping is deterministic:

> For all e, if $e \mapsto v$ and $e \mapsto v'$ then $v =_\alpha v'$ (v and v' are equal values, where equal means α-equivalent).

Determinism of Multi-step Evaluation Define multi-step evaluation as the *reflexive-transitive closure* of evaluation:

$$\frac{}{e \mapsto e} \qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}$$

Task 2.2. Prove that multi-stepping is deterministic:

> For all e, if $e \mapsto^* v$ and $e \mapsto^* v'$ then $v =_\alpha v'$ (v and v' are equal values, where equal means α-equivalent).

Relating SOS and Evaluation One way to define "evaluating a program all the way to a value" is defining structural operational semantics $e \mapsto e'$, and then defining multistep evaluation as $e \mapsto^* e'$. Another is to directly define an *evaluation relation* $e \Downarrow v$. For example, for call-by-name we have the following rules for $\to$:

$$\frac{}{(\lambda x : \tau.e) \Downarrow (\lambda x : \tau.e)} \qquad \frac{e_1 \Downarrow (\lambda x : \tau.e) \qquad e[e_2/x] \Downarrow v}{(e_1\, e_2) \Downarrow v}$$

Task 2.3. Prove that the evaluation relation and multi-step SOS correspond:

$e \Downarrow v$ iff $e \mapsto^* v$

You will need some lemmas; if you get stuck, see PFPL Section 7.2 for a hint.

3 Progress and Preservation

If you haven't seen progress and preservation proofs before, first read the proof for functions and application in PFPL section 8.2.

Natural Numbers **Task 3.1**. Prove progress and preservation for the natural numbers in Gödel's T, i.e. do the cases of progress and preservation for 0, successor, and the recursor (recursion).

Products Read PFPL Section 10.1, and then and prove Theorem 10.1, progress and preservation for a language with pair types $\tau_1 \times \tau_2$.

Task 3.2. Prove preservation, that is:

If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Task 3.3. Prove progress. That is:

If $\cdot \vdash e : \tau$ then either e val or there is some e' such that $e \mapsto e'$.

Sums **Task 3.4**. Read PFPL Section 11.1 and prove Theorem 11.1, progress and preservation for the sum type $\tau_1 + \tau_2$.

4 Logical Relations

In lecture, we defined a logical relation and used it to prove termination. The relation $\mathsf{Good}_\tau(e)$ is a predicate on closed terms $\cdot \vdash e : \tau$, and is defined by induction on τ as

- $\mathsf{Good}_{\mathtt{int}}(e)$ iff there exists a number k such that $e \mapsto^* \mathsf{num}[k]$
- $\mathsf{Good}_{\tau_2 \to \tau}(f)$ iff there exists a term $x : \tau_2 \vdash e : \tau$ such that (1) $f \mapsto^* (\lambda x.e)$ and (2) for all $\cdot \vdash e_2 : \tau_2$, $\mathsf{Good}_{\tau_2}(e_2)$ implies $\mathsf{Good}_\tau(e[e_2/x])$.

A substitution $\cdot \vdash \theta : \Gamma$ is a list $(e_1/x_1, \ldots e_n/x_n)$ where $\cdot \vdash e_i : \tau_i$ for each variable $x_i : \tau_i$ in Γ. We write $e[\theta]$ for performing the substitution θ, plugging in for all of the variables in e. For a substitution $\cdot \vdash \theta : \Gamma$, we say $\mathsf{Good}_\Gamma(\theta)$ iff $\mathsf{Good}_\tau(\theta(x))$ for all $x : \tau$ in Γ.

The fundamental theorem says that well-typed terms preserve goodness:

If $\Gamma \vdash e : \tau$ and $\mathsf{Good}_\Gamma(\theta)$ then $\mathsf{Good}_\tau(e[\theta])$.

Task 4.1. Extend the languages with binary pairs as in PFPL Section 10.1 with a lazy operational semantics. This means that $\langle e_1, e_2 \rangle$ is always a value, and does not evaluate e_1 and e_2 until a projection is taken. Check that closure under converse evaluation still holds, and do the cases of the fundamental theorem for pairing and projections.

Task 4.2. Extend the definition of the logical relation to sum types $\tau_1 + \tau_2$, again with a lazy operational semantics (so both $\mathsf{inl}(e)$ and $\mathsf{inr}(e)$ are values regardless of whether e is). Check that closure under converse evaluation still holds, and do the cases of the fundamental theorem for injections and case analysis.

Task 4.3. Replace the base type `int` with natural numbers as in System T (again with a lazy successor, so $\mathtt{s}(e)$ is always a value). Define the logical relation for `nat` by an inner or "horizontal" induction:

$$\frac{e \mapsto^* \mathtt{z}}{\mathsf{Good}_{\mathtt{nat}}(e)} \qquad \frac{e \mapsto^* \mathtt{s}(e_0) \quad \mathsf{Good}_{\mathtt{nat}}(e_0)}{\mathsf{Good}_{\mathtt{nat}}(e)}$$

You will need to do an inner induction on $\mathsf{Good}_{\mathtt{nat}}(e)$ to prove the case of the fundamental theorem for the recursor.

Task 4.4. Adapt all of the above to call-by-value/eager instead of call-by-name/lazy. What changes in the proofs?

5 Implementing System T with Labeled Sums and Products

For this section of the assignment, we will be working on System T extended with labeled sum and product types. The abstract syntax is as follows; on the left is the formal abstract binding tree notation, and on the right is a more reader-friendly display notation, which should be thought of as synonyms for the left-hand column.

Type	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$	naturals
			$\mathtt{arr}(\tau_1; \tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{prod}(l_1 \hookrightarrow \tau_1; \ldots; l_n \hookrightarrow \tau_n)$	$\langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle$	product
			$\mathtt{sum}(l_1 \hookrightarrow \tau_1; \ldots; l_n \hookrightarrow \tau_n)$	$[l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n]$	sum
Exp	e	$::=$	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{rec}\{e_z; x.y.e_1\}(e)$	$\mathtt{rec}(e)\{\mathtt{z} \hookrightarrow e_z \mid \mathtt{s}(x)\ \mathtt{with}\ y \hookrightarrow e_1\}$	recursion
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda(x : \tau)\ e$	abstraction
			$\mathtt{ap}(e_1; e_2)$	$e_1(e_2)$	application
			$\mathtt{tuple}(l_1 \hookrightarrow e_1; \ldots; l_n \hookrightarrow e_n)$	$\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle$	tuple
			$\mathtt{proj}[l](e)$	$e \cdot l$	projection
			$\mathtt{in}[l]\{\tau\}(e)$	$l \cdot e$	injection
			$\mathtt{case}[l_1, \ldots, l_n](x_1.e_1; \ldots; x_n.e_n)(e)$	$\mathtt{case}\ e\ \mathtt{of}\ \{l_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid l_n \cdot x_n \hookrightarrow e_n\}$	casing

Labels, represented by l in the grammar, are simply identifiers associated with elements of tuples and sums to give them some description and allow for a more intuitive elimination form.

The types of this language are natural numbers and function types, together with labeled products and and labeled sums.

A labeled product is a *record* with named fields. To create a record, you give a value for each field. To use a record, you project a specified field. Records are included in some form or another in most functional languages (e.g. ML, Haskell). Records are somewhat analogous to the fields of an object in object-oriented programming languages; additionally, many languages have a notion of a function with named arguments, which can be modeled by a function that takes a record as an argument.

More formally, the type $\langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle$ is a labeled product type whose component labeled l_i has type τ_i. The introduction form for labeled products has the form $\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle$ whose l_i component is e_i. The projection $e \cdot l$ selects the l component of the labeled tuple e.

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n \qquad \forall i \neq j . l_i \neq l_j}{\Gamma \vdash \langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle : \langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle} (\times\text{-I})$$

$$\frac{\Gamma \vdash e : \langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle \qquad (1 \leq i \leq n)}{\Gamma \vdash e \cdot l_i : \tau_i} (\times\text{-E})$$

$$\frac{e_1 \ \mathsf{val} \ \ldots \ e_n \ \mathsf{val}}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \ \mathsf{val}} (\mathtt{tuple_v}) \qquad \frac{e_1 \ \mathsf{val} \ldots e_{i-1} \ \mathsf{val} \qquad e_i \mapsto e_i'}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \mapsto \langle l_1 \hookrightarrow e_1, \ldots, l_i \hookrightarrow e_i', \ldots, l_n \hookrightarrow e_n \rangle} (\mathtt{tuple_s})$$

$$\frac{e \mapsto e'}{e.l_i\{\tau\} \mapsto e'.l_i\{\tau\}} (\mathtt{proj}_\mathtt{s}^1) \qquad \frac{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \ \mathsf{val}}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \cdot l_i \mapsto e_i} (\mathtt{proj}_\mathtt{s}^2)$$

Dually, a labeled sum type is a like a non-recursive datatype in ML or Haskell. To make a value of this type, you choose a label and supply the appropriate payload data. To use a value of this type, you case-analyze the possible ways it could have been constructed. The type $[l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n]$ is the labeled sum type whose elements are elements of type τ_i labeled with l_i, for some $1 \leq i \leq n$. The injection $\mathtt{inj}\{\tau\}(l; e)$ creates a sum of type τ with e at label l. $\mathtt{case}\ e\ \mathtt{of}\ \{l_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid l_n \cdot x_n \hookrightarrow e_n\}$ takes a sum and if it is of form l_i, binds the contents to x_i in e_i.

$$\frac{\Gamma \vdash e : \tau_i \qquad \forall i \neq j . l_i \neq l_j}{\Gamma \vdash l_i \cdot e : [l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n]} (+\text{-I})$$

$$\frac{\Gamma \vdash e : [l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n] \qquad \Gamma, x_i : \tau_i \vdash e_i : \tau \ \ldots \ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \{l_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid l_n \cdot x_n \hookrightarrow e_n\} : \tau} (+\text{-E})$$

$$\frac{e \ \mathsf{val}}{l \cdot e \ \mathsf{val}} (\mathtt{inj_v}) \qquad \frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'} (\mathtt{inj_s})$$

$$\frac{e \mapsto e'}{\mathtt{case}\ e\ \mathtt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\} \mapsto \mathtt{case}\ e'\ \mathtt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\}} (\mathtt{case}_\mathtt{s}^1)$$

$$\frac{l_i \cdot e \ \mathsf{val}}{\mathtt{case}\ l_i \cdot e\ \mathtt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\} \mapsto [e/x_i]e_i} (\mathtt{case}_\mathtt{s}^2)$$

Your goal in this section will be to implement the static and dynamic semantics of System T with labeled sums and products. The full rules are included in Section 6 for your reference.

You are welcome to do this in any language if you want to start from scratch. However, if you use Standard ML, we have provided a lot of definitions and scaffolding that you will find useful.

Data Structures The directory `cmlib` contains a general SML library. You may want to use some operations from the signatures in `dict.sig` and `set.sig` for dictionaries and sets.

Names One common issue in implementing programming languages is representing names/variables/identifiers in such a way that you can check α-equivalence of syntax trees and correctly implement (capture-avoiding) substitution.

In the `var` subdirectory, we have provided an abstract type of of "temp"oraries/names, which are used to implement variables and labels. The file `var/temp.sig` describes the operations on names. Key operations include:

```
signature TEMP =
sig
  type t
  (* Creates a new, globally unique temp. *)
  val new : string -> t
  (* Tests whether two temps are equal. *)
  val equal  : (t * t) -> bool
  ...
end
```

The type `t` is an abstract/private/unknown type with the operations provided in the rest of the signature. The operation `new` creates a new *globally unique* name (i.e. every time you call `new`, you are guaranteed that the name provided will be different than any other name that already exists). Though it is not necessary (e.g. you could just as well always call `new ""`, because each new name is different), for readability, `new` takes a string that is used when the name is printed out. Names can be compared for equality using the `equal` function.

In your implementation, the modules Label and Variable have signature `TEMP`.

Abstract binding trees Next, we have implemented the syntax of types and terms; see `labT/labt.sig` for the interface.

As a first cut, you might expect that we would represent terms with a datatype like

```
datatype term
  = Var of termVar
  | Lam of (termVar * Typ.t) * term
  | Ap of term * term
```

where `termVar` is a type of names as described above. This says that a `Lam` term has a variable, a type ascription for the variable, and another term standing for the body of the function; an application has two subterms, for the function and the argument. A problem with this representation is that, as you traverse a term, you need to remember to substitute fresh variables at appropriate times to avoid accidental problems with α-equivalence and substitution, because the same `termVar` might be used in many places with different meanings.

Instead, we will use an abstract type to hide all of this variable manipulation in the implementation of terms. Here is the signature:

```
structure Term : sig
  type termVar = Variable.t
  type term

  datatype view
    = Var of termVar
    | Z
    | S of term
    | Lam of (termVar * Typ.t) * term
    | Ap of term * term
    | Pair of (Label.t * term) list
    | Proj of term * Label.t
    | Inj of Typ.t * Label.t * term
    | Case of term * ((Label.t * termVar) * term) list
    | Iter of term * term * (termVar * (termVar * term))

  val into : view -> term
  val out : term -> view

  val aequiv : term * term -> bool
  val subst : term -> termVar -> term -> term

  ...

end
```

The type `term` is abstract, while the type `view` is a datatype like above—but not that the subtrees are abstract `term`s, not recursive occurences of `view`. There are functions `into` and `out` that map between these, so to *create* a term, you can call `into` on a `view`, while to case-analyze a term (as you will do in your type checker and operational semantics), you call `out`. The key idea is:

> *whenever `out` exposes a `termVar` in the view (e.g. in a `Lam` or `Case` or `Iter`), that variable is* fresh *(distinct from all other variables in play)*

Therefore, in your code, you can assume all variables are fresh/distinct, which simplifies the implementation.

For conveneince, the `Term` module also provides functions named `Var'` and `Z'` and `S'` and `Lam' ...`, where `C'` abbreviates `into(C ...)` for the corresponding view constructor.

The `Term` module also provides implementations of α-equivalence and substitution for your convenience.

Your code Your code will go in the `sem` directory, in `typechecker.sml` and `dynamics.sml`, which implement the signatures in the corresponding `.sig` files.

Task 5.1. Implement the statics of the language. You will need to implement the module `TypeChecker`, which can be found in "`sem/typechecker.sml`".

Hints:

- Use the structure `Context` for dictionaries mapping term variables to types.
- Your main goal is to implement the function `checkType`, which takes a context and a term and *synthesizes* a type if the expression is well-typed. If the expression is ill-typed, the function should raise the exception `TypeError`.
- The helper function `equiv e t1 t2` checks if the types `t1` and `t2` are equal. If so, it returns `()`, the one value of the `unit` type, and if not, it raises a `TypeError` signalling that there was a problem type checking `e` (which is provided only for error reporting). A common SML idiom is

  ```
  let val () = equiv e t1 t2
   in rest
   end
  ```

  which signals that `equiv e t1 t2` is being run only for its computational effect of possibly raising an exception, and when it succeeds, we run `rest`. This can also be written

  ```
  equiv e t1 t2; rest
  ```

- Your type checker should ensure that all types occuring in type checking are well-formed, which in this case means only that the labeled sums and products have no duplicate labels.

Task 5.2. Implement the dynamics of the language. You will need to implement the module `LabTDynamics`, which can be found in "`sem/dynamics.sml`".

Your overall goal is to implement the function

```
datatype d = STEP of Term.term | VAL
val trystep : Term.term -> d
```

(the rest of the module is implemented for you). This should satisfy the following specification:

> If `e` is a closed, well-typed term, then `trystep e` returns `STEP e'` if $\texttt{e} \mapsto \texttt{e}'$, or `VAL` if `e` is a value.

Hints:

- `trystep` is an *implementation of the progress proof for the language*. If you get stuck on how to proceed, try doing the corresponding case of the proof.
- Ignore the exceptions `RuntimeError` and `Abort` and the type `D`, which are not used here (they are for other uses of this support code).
- Because we are *assuming* and `e` is well-typed (i.e. we run the type checker before running the operational semantics), your implementation can do anything it wants if it encounters an ill-typed term—there is no need to specifically test for such cases and report them. However, we have provided an exception `Malformed` that could be used in cases where you do discover that the term is ill-typed.

Testing To compile your program, start `sml` from the handout directory and then do

```
- CM.make "sources.cm";
```

Our support code provides a *read-typecheck-eval-print-loop* (REPL) that you can use to test interactively. To start it, run

```
- TopLevel.repl()
```

The REPL keeps tract of a current expression that you are working. There are three commands that you can use

- `load e;`. Type checks the expression `e` and sets it as the current expression to be evaluated.
- `step;` steps the current expression one step, and reports the new expression or if the current expression is a value.
- `eval;` evaluates the current expression all the way to a value.

The commands `step` and `eval` can also be called with an expression argument, which is like running `load e` before them.

Here are some examples that show the concrete syntax (more examples are in the `tests` directory):

1. ```
   ->load s(z);
   Statics: term has type Nat
    |--> (S Z)
   ->step;
    (S Z) VAL
   ->step;
   Error: Nothing to step!
   ```

2. ```
   ->eval fn (x:nat) x;
   Statics: term has type (Arrow (Nat, Nat))
   Statics: term has type (Arrow (Nat, Nat))
    (Lam ((x@13, Nat) . x@13)) VAL
   ```

3. ```
   ->load (fn (x:nat) x) z;
   Statics: term has type Nat
    |--> (Ap ((Lam ((x@18, Nat) . x@18)), Z))
   ->step;
   Statics: term has type Nat
    |--> Z
   ->step;
    Z VAL
   ```

4. Type errors should be reported:

   ```
   ->load s(fn (x:nat) x);
   LabTChecker error in term: (S (Lam ((x@38, Nat) . x@38)))
   ```
   ```

5. $\texttt{rec}(e)\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x)\ \texttt{with}\ p \hookrightarrow e_1\}$ is written `iter e { e0 | s(x) with p => e1}`.

```
->eval (fn (x:nat) iter x { z | s(y) with p => s(s(p))}) (s(s(z)));
Statics: term has type Nat
Statics: term has type Nat
 (S (S (S (S Z)))) VAL
```

6. Record values are written `< l1 = e1 , ... , ln = en>`.

```
->eval <bar = z, foo = s(z)>.foo;
Statics: term has type Nat
Statics: term has type Nat
 (S Z) VAL
```

7. Because types are inferred, sum injections must be annotated with all of the constructors (a better type checker would infer these). For example, what we would write as $\mathsf{SOME} \cdot e$ in the labeled sum type

$$[\mathsf{NONE} \hookrightarrow \langle\rangle, \mathsf{SOME} \hookrightarrow \texttt{nat}]$$

is written

```
in[NONE :: <>, SOME :: nat]{SOME}(z)
```

8.
```
->eval case (in[NONE :: <>, SOME :: nat]{SOME}(z))
=>          {NONE m => z | SOME w => s(z)};
Statics: term has type Nat
Statics: term has type Nat
 (S Z) VAL
```

You can also run an example from a file using `TopLevel.evalFile filename`, and there is an automated testing harness in `tests.sml`.

6 Appendix: Full Rules for Gödel's T with labeled products and sums

6.1 Statics

6.1.1 System T

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}(\text{var})$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}}(\texttt{nat-I}_1) \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}}(\texttt{nat-I}_2) \qquad \frac{\Gamma \vdash e : \texttt{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}(e)\{\texttt{z} \hookrightarrow e_z \mid \texttt{s}(x)\ \texttt{with}\ y \hookrightarrow e_1\} : \tau}(\texttt{rec})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \tau_1 \to \tau_2}(\to\text{-I}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}(\to\text{-E})$$

6.1.2 Labeled Sums and Products

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n \qquad \forall i \neq j . l_i \neq l_j}{\Gamma \vdash \langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle : \langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle} (\times\text{-I}) \qquad \frac{\Gamma \vdash e : \langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle \qquad (1 \leq i \leq n)}{\Gamma \vdash e \cdot l_i : \tau_i} (\times\text{-E})$$

$$\frac{\Gamma \vdash e : \tau_i \qquad \forall i \neq j . l_i \neq l_j}{\Gamma \vdash l_i \cdot e : [l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n]} (+\text{-I})$$

$$\frac{\Gamma \vdash e : [l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n] \qquad \Gamma, x_i : \tau_i \vdash e_i : \tau \ \ldots \ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{l_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid l_n \cdot x_n \hookrightarrow e_n\} : \tau} (+\text{-E})$$

6.2 Dynamics (Eager, Left-to-Right)

6.2.1 System T

$$\frac{}{\texttt{z}\ \mathsf{val}} (\texttt{nat}^1_{\mathsf{v}}) \qquad \frac{e\ \mathsf{val}}{\texttt{s}(e)\ \mathsf{val}} (\texttt{nat}^2_{\mathsf{v}}) \qquad \frac{}{\texttt{lam}[\tau](x.e)\ \mathsf{val}} (\rightarrow_{\mathsf{v}}) \qquad \frac{e \mapsto e'}{\texttt{s}(e) \mapsto \texttt{s}(e')} (\texttt{nat}_{\mathsf{s}}) \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} (\text{ap}^1_{\mathsf{s}})$$

$$\frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'} (\text{ap}^2_{\mathsf{s}}) \qquad \frac{e_2\ \mathsf{val}}{\texttt{lam}[\tau](x.e)\ e_2 \mapsto [e_2/x]e} (\text{ap}_{\mathsf{e}}) \qquad \frac{e \mapsto e'}{\texttt{rec}(e_z)\{\texttt{z} \hookrightarrow x \mid \texttt{s}(y)\ \texttt{with}\ e_1 \hookrightarrow e\}} (\texttt{rec}^1_{\mathsf{s}})$$

$$\frac{}{\texttt{rec}(e_z)\{\texttt{z} \hookrightarrow x \mid \texttt{s}(y)\ \texttt{with}\ e_1 \hookrightarrow z\} \mapsto e_z} (\texttt{z})(\texttt{rec}^2_{\mathsf{s}})$$

$$\frac{e\ \mathsf{val}}{\texttt{rec}(e_z)\{\texttt{z} \hookrightarrow x \mid \texttt{s}(y)\ \texttt{with}\ e_1 \hookrightarrow \texttt{s}(e)\} \mapsto [e, \texttt{rec}(e_z)\{\texttt{z} \hookrightarrow x \mid \texttt{s}(y)\ \texttt{with}\ e_1 \hookrightarrow e\}/x, y]e_1} (\texttt{rec}^3_{\mathsf{s}})$$

6.2.2 Labeled Products and Sums

$$\frac{e\ \mathsf{val}}{l \cdot e\ \mathsf{val}} (\texttt{inj}_{\mathsf{v}}) \qquad \frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'} (\texttt{inj}_{\mathsf{s}})$$

$$\frac{e \mapsto e'}{\texttt{case}\ e\ \texttt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\} \mapsto \texttt{case}\ e'\ \texttt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\}} (\texttt{case}^1_{\mathsf{s}})$$

$$\frac{l_i \cdot e\ \mathsf{val}}{\texttt{case}\ l_i \cdot e\ \texttt{of}\ \{l_1(x_1) \hookrightarrow e_1 \mid \ldots \mid l_n(x_n) \hookrightarrow e_n\} \mapsto [e/x_i]e_i} (\texttt{case}^2_{\mathsf{s}})$$

$$\frac{e_1\ \mathsf{val}\ \ldots\ e_n\ \mathsf{val}}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle\ \mathsf{val}} (\texttt{tuple}_{\mathsf{v}}) \qquad \frac{e_1\ \mathsf{val} \ldots e_{i-1}\ \mathsf{val} \qquad e_i \mapsto e_i'}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \mapsto \langle l_1 \hookrightarrow e_1, \ldots, l_i \hookrightarrow e_i', \ldots, l_n \hookrightarrow e_n \rangle} (\texttt{tuple}_{\mathsf{s}})$$

$$\frac{e \mapsto e'}{e.l_i\{\tau\} \mapsto e'.l_i\{\tau\}} (\texttt{proj}^1_{\mathsf{s}}) \qquad \frac{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle\ \mathsf{val}}{\langle l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n \rangle \cdot l_i \mapsto e_i} (\texttt{proj}^2_{\mathsf{s}})$$