Assignment #1, Update 2:
Abstract Binding Trees, Dynamics and Statics

15-312: Principles of Programming Languages

Out: Tuesday, January 21st, 2014
Due: Tuesday, February 4th, 2014 1:29PM

Introduction

This assignment consists of two major parts. The first part involves building up an infrastructure to represent and manipulate abstract syntax with binding. The second part will be to implement the calculator language we have been discussing in class, also introduced in Ch. 4 of PFPL. You will begin by implementing a version without a static type system, then you will implement a statically typed version to see the key differences between these approaches.

There is quite a bit of code to understand and implement in this assignment. Start early.

0.1 Submission

We will collect *exactly* the following files:

```
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/assn1.pdf
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/abt.sml
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/abt-util.sml
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/termops.sml
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/checkeddynamics.sml
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/typechecker.sml
/afs/andrew/course/15/312/handin/<yourandrewid>/assn1/uncheckeddynamics.sml
```

Make sure that your files have the right names (especially `assn1.pdf`!) and are in the correct directory.

1 Abstract Binding Trees

Abstract binding trees, introduced in Section 1.2 of PFPL, are used to represent the abstract syntax of languages with variables and binding. We will begin by building a generalized infrastructure to support abstract binding trees for any such language.

1.1 Variables

A variable in an abstract binding tree (ABT) is a placeholder for a fixed, but unspecified ABT. A key idea is the notion of a *new* or *fresh variable* – one that has not been seen before in some given context. To work with variables, we introduce the `VARIABLE` signature, reproduced below.

There are several ways to guarantee uniqueness, but one simple way is to just ignore the context and make sure every freshly generated variable is globally unique. We provide such an implementation that satisfies this signature for you in the `Var` module. You can read the source code in `var.sml` if you'd like, but you don't have to (that's the magic of signatures!)

1.2 Operators

In PFPL, we specify every language as a collection of operators. Each operator o has an arity $\mathsf{ar}(o) = (n_1, \ldots, n_k)$, where the length k gives the number of arguments the operator takes and each n_i specifies the *valence* of the i-th argument, which is the number of variables bound by that argument.

We will work with operators via a signature, `OPERATOR`.

```
signature OPERATOR =
sig
  type t

  val arity : t -> int list
  val equal : (t * t) -> bool
  val toString : t -> string
end
```

We will ask you to give an implementation of this signature in Section 2 below. But for now, what we're going to do is develop an implementation of ABTs that is parametric in the `OPERATOR` signature, so that you can instantiate it in different ways to automatically construct the syntax for the different languages we will study in this course.

1.3 Abstract Binding Trees

Now, we can introduce the signature of abstract binding trees. This is a version of an interface to syntax that has been developed here at CMU over numerous compiler development efforts, and is designed to help users avoid running into some very common errors.

```
signature ABT =
sig
   structure Variable : VARIABLE
   structure Operator : OPERATOR

   type t

   datatype 'a view =
     ` of Variable.t
```

```
  | \ of Variable.t * 'a
  | $ of Operator.t * 'a list

  exception Malformed

  val into : t view -> t
  val out : t -> t view

  val aequiv : t * t -> bool
  val map : ('a -> 'b) -> 'a view -> 'b view
end
```

The first two fields of the `ABT` signature simply say that it has two sub-modules, `Variable` and `Operator`, which are the variable and operator implementations for the particular language syntax being implemented.

Then, we introduce an abstract type `t` for the actual internal represention of an abstract binding tree. However, there are no simple functions that construct values of type `t` directly. Instead, access to the representation type `t` is mediated by a *view*, which is a **non-recursive** datatype `'a view`, and a pair of functions `into` and `out`. The idea behind a view is that it is a type whose values represent a one-step unfolding of a value of the abstract type `t`. The function `out` does this one-step unfolding of an ABT to create a view, and `into` puts a view back together into an ABT.

More specifically, the function `out` has type `t -> t view`. This means it takes an ABT value, and deconstructs it as follows:

- Suppose that the term `e` of type `t` represents the abstract binding tree x. Then, a call `out e` would return `` ` x ``, where the value `x`, of type `Variable.t`, represents the variable x.
- Likewise, suppose a term `e` represents an abstract binding tree $f(e_1, \ldots, e_n)$, where f is an operator. Then a call `out e` should return `$ (f, es)`, where `f` is a term of type `Operator.t` representing f, and `es` is a term of type `t list`, whose elements represent the ABTs of each e_i in the argument sequence $(e_1, \ldots, e_n)$ – but *not* views to them. This is what we mean by a one-step unfolding.
- Finally, if a term `e` represents an abstract binding tree $x.e'$, then the call `out e` will return `\ (y, e'')` where `y` represents a fresh variable y and `e''` represents the ABT $[y \leftrightarrow x]e'$, where $[y \leftrightarrow x]e$ means "substitute *free* occurrences of x in e with y".

  In other words, the `out` operation renames the variable bound by an abstractor whenever it unpacks it! This is one of the essential features of this interface, and implementing this correctly will save you many hours tracing down strange α-conversion issues.

The function `into` has type `t view -> t`, and folds a one-step unfolding (that is, a view) back into an ABT term.

- Suppose `x` is a term of type `Variable.t`, which represents the variable x. Then `into (` `` ` x ``)` will return the abstract binding tree corresponding to the variable occurence x.

- If `x` is a term of type `Variable.t`, which represents the variable x; and `e` is a term of type `t` representing the ABT e, then `into (\ (x, e))` will return a term representing the abstractor $x.e$.
- If `f` is a term of type `Operator.t`, representing the operator f, and `es` is a term of type `t list`, representing the sequence $(e_1, \ldots, e_n)$. Then `into ($ (f, es))` will return a term representing $f(e_1, \ldots, e_n)$ if the arity of f matches the sequence. Otherwise, it will raise the exception `Malformed`.

However, `into` is not quite the inverse of `out`. If `e` is a term of type `t`, then `e` and `into(out(e))` will not in general represent equal ABTs – but they will, however, always represent α-equivalent ABTs! We can test α-equivalence with the `aequiv` function, which returns true if its two arguments are α-equivalent, and false otherwise. This means that `aequiv(into(out(e)), e)` is always true.

Finally, we have a convenience function `map` in this interface. Given a function `f` of type `'a -> 'b`, it takes an `'a view` and produces a `'b view` by applying it to each subterm of type `'a`, and leaving all of the other subterms and constructors unchanged.

Task 1 (5%). Suppose `v` has type `t view`. Will `out(into(v))` be equal to `v`? Explain why or why not.

Think carefully about what effect the renaming `out` can perform before answering this question.

1.4 Implementing ABTs

One of the inconveniences of using a straightforward representation of ABTs is that α-equivalent terms can have multiple representations, so implementing `aequiv` and other helper functions becomes tricky. We will look at a more sophisticated representation, called *locally nameless form*, which avoids this problem, so that each α-equivalence class is represented with a single data value, and thus α-equivalence can be tested for with a simple structural traversal.

The basic idea is to observe that variables in an abstract binding tree serve two roles. First, they can appear free in a ABT – that is, they can be a *name* for a hypothesis in some hypothetical context. Second, they can be a bound variable, in which case the variable occurences *refer back* to the location of the binding abstraction. For example, in the ML term

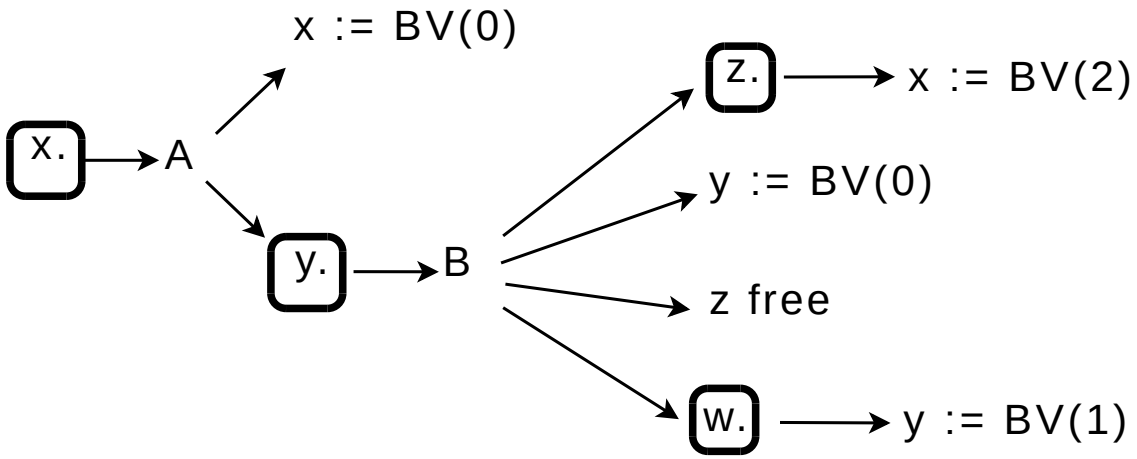
```
let x = 5 in let y = 3 in (x, x + y + C)
```

The bound variable `x` refers back to the outermost binding, and the bound variable `y` refers back to the second binding. The variable `C` is a free variable. The only reason we need the names of the bound variables is to distinguish one abstraction site from another – the names themselves are irrelevant from a semantic point-of-view. (This is just another way of saying that we want to identify terms up to α-equivalence, of course.)

In a locally nameless representation, we distinguish these two roles in our data structure, in order to make α-equivalence implementable as structural equality on terms. The trick in this representation is to exploit a structural invariant of binding trees. First, binding trees are *trees* – they have no cycles in them. Second, in any abstraction $x.e$, the only occurences of x are within e. As a result of these two facts, we have a *unique* path from an abstractor to each occurence of the variable it binds. Furthermore, since we're only interested in the binder sites, we can compress

this path to a single number, which tells us how many binders we have to hop over before we reach the one we're interested in.

Consider the following diagram of the binding tree $x.A(x, y.B(z.x, y, z, w.y))$, in which A and B are operator names.



We've put a box around every abstraction, and labelled each bound variable with its bound variable number. We can calculate the bound variable number by looking at each path from an abstraction to its use sites, and count the number of abstractions crossed along the way:

Path	Variable #
$\boxed{x.} \to A \to x$	0
$\boxed{x.} \to A \to \boxed{y.} \to B \to \boxed{z.} \to x$	2
$\boxed{y.} \to B \to y$	0
$\boxed{y.} \to B \to \boxed{w.} \to y$	1

An important fact to notice about these paths is that even for the same binder, each occurrence of its bound variable can have a *different* bound variable number, depending on the number of abstractions we crossed over to reach that variable occurrence.

Task 2 (20%). You will need to implement the structure Abt in file `abt.sml`, implementing ABTs with a locally nameless representation:

```
functor Abt(O : OPERATOR) :> ABT where type Variable.t = Var.t
                                 where type Operator.t = O.t
=
struct
   open List_Util

   structure Variable = Var
```

```
  structure Operator = O

  datatype 'a view =
    ` of Variable.t
  | \ of Variable.t * 'a
  | $ of Operator.t * 'a list

  datatype t =
    FV of Var.t
  | BV of int
  | ABS of t
  | OPER of Operator.t * t list

  exception Malformed
  exception NotImplemented

  (* ... your solution goes here ... *)
  fun into x    = raise NotImplemented
  fun out x     = raise NotImplemented
  fun aequiv x  = raise NotImplemented
  fun map x y   = raise NotImplemented

end
```

Hint: When implementing this structure, you will find it helpful to define two functions `bind` and `unbind`.

- `bind` should take a term e and a free variable x, and return a new term corresponding to $x.\ e$.
- `unbind` should take a term $x.\ e$, and return a variable x', and a term corresponding to $[x'/x]e$.

Bonus hint: Note that one of the invariants of the locally nameless representation is that the case `BV n` only occurs once you've gone beneath a binder. So it can't happen at the top level of a term.

Bonus bonus re-hint: Your implementation of `aequiv` is a structural equality comparison.

1.4.1 Implementing Substitution and Finding Free Variables

So far, the ABT interface you've implemented has been very spartan. It doesn't even sport a function to do substitution.

However, the reason you haven't been asked to put that into the basic interface is that it can be implemented using the functionality that you've already developed.

Task 3 (10%). So now you need to implement a functor `ABT_Util`:

```
functor ABT_Util(A : ABT) : ABT_UTIL =
struct
```

```
  open A

  (* ... your solution goes here ... *)
   exception NotImplemented

   fun freevars e     = raise NotImplemented
   fun subst a x b    = raise NotImplemented
   fun `` e           = raise NotImplemented
   fun \\ e           = raise NotImplemented
   fun $$ e           = raise NotImplemented
   fun toString e     = raise NotImplemented
end
```

which implements the `ABT_UTIL` signature:

```
signature ABT_UTIL =
sig
  include ABT

  val freevars : t -> Variable.t list
  val subst    : t -> Variable.t -> t -> t

  val `` : Variable.t -> t
  val \\ : Variable.t * t -> t
  val $$ : Operator.t * t list -> t

  val toString : t -> string
end
```

The specifications of these operations are:

- `freevars e` should return a list containing the free variables in `e`, with each variable occurring at most once. (That is, the same variable should not be repeated.)
- Suppose that `e1` represents e_1, `x` represents x, and `e2` represents e_2. Then `subst e1 x e2` should return a new term corresponding representing the result of performing the substitution $[e_1/x]e_2$,
- The ``, \\, and `$$` functions are convenience functions (which we won't test) used to form terms of type `ABT.t` directly (simply skipping the need to call `into`),
- `toString` should return a human-readable representation of an ABT. Once again, we won't test this function – it will just be useful for your own debugging purposes.

2 Language

Now we have all the infrastructure we need to implement a language. We will be working with the language $\mathcal{L}\{\texttt{num str}\}$, defined in Ch. 4 of *PFPL*.

Task 4 (5%). Implement the expression operators of $\mathcal{L}\{\texttt{num str}\}$ using a structure `TermOps`. The implementation has been partially provided for you in `termops.sml`.

```
structure TermOps =
struct

  datatype t = Num of int | Str of string | Plus | Times | Cat | Len | Let

  (* ... your solution goes here ... *)
   exception NotImplemented
   fun arity e = raise NotImplemented
   fun equal e = raise NotImplemented

  fun toString (Num n) = Int.toString n
    | toString (Str s) = "\""^s^"\""
    | toString Plus = "plus"
    | toString Times = "times"
    | toString Cat = "cat"
    | toString Len = "len"
    | toString Let = "let"
end
```

With this structure, we can now apply the functors we defined above to get an ABT structure for terms of our language, as shown in `term.sml`:

```
structure Term = ABT_Util(Abt(TermOps))
```

2.1 Dynamics

Let's now implement a dynamics for these terms. For reference, the rules describing the dynamics of $\mathcal{L}\{\texttt{num str}\}$ are given in Appendix A below.

We will use the following signature to implement our dynamics:

```
signature DYNAMICS =
sig
  type d

  datatype D = Step of Term.t | Val | Err
  val view : d -> D

  exception Malformed

  val trystep : Term.t -> d

  exception RuntimeError

  val eval : Term.t -> Term.t
end
```

The `trystep` function takes a closed term and produces a value of type `d`, which is an abstract type representing which judgement of the three defined in Appendix A applies for the term. Values of this abstract type can be case analyzed via the `view` function, which converts it to a value of the datatype `D`. (We use an abstract type `d` combined with a view rather than `D` directly for reasons that will become clear below.) The `Malformed` exception can be raised by `trystep` for malformed terms (e.g. open terms, binders outside of operators, arity mismatches, etc.).

The `eval` function should fully evaluate the provided term to a value (i.e. take all the steps), or raise a `RuntimeError` if this cannot be done.

Task 5 (20%). Implement the full dynamics of $\mathcal{L}\{\texttt{num str}\}$ as defined in Appendix A in the file `checkeddynamics.sml`.

```
structure CheckedDynamics : DYNAMICS =
struct

  exception RuntimeError
  exception Malformed
  exception NotImplemented

  datatype D = Step of Term.t | Val | Err

  (* ... your solution goes here ... *)
  datatype d = ToBeImplemented

  fun view e        = raise NotImplemented
  fun trystep e     = raise NotImplemented
  fun eval e        = raise NotImplemented
end
```

2.2 Statics

There is a lot of error checking going on in the dynamics in Appendix A. But as we've been discussing in class, we can eliminate much or all of this by equipping our language with a static type system (see Ch. 6 of PFPL)! The type checking rules for $\mathcal{L}\{\texttt{num str}\}$ are reproduced in Appendix B for your reference.

Task 6 (10%). (Unicity of Typing) Prove that for every typing context Γ and expression e, there exists at most one τ such that $\Gamma \vdash e : \tau$.

Update: You can do this for the sublanguage that is missing the `times` *and* `cat` *syntax, the two corresponding typing rules, the six corresponding evaluation rules, and the eight corresponding error rules.*

Task 7 (10%). (Canonical Forms) Prove that if e val and $e : \tau$, then

1. if $\tau = \texttt{num}$ then $e = \texttt{num}[n]$ for some number n.
2. if $\tau = \texttt{str}$ then $e = \texttt{str}[s]$ for some string s.

Update: Please do this for the full language.

2.2.1 Implementing Type Checking

We can represent types using the ABT infrastructure that we developed above as well. In particular, each type can be considered a type operator with no arguments. This is implemented in the `typeops.sml` file:

```
structure TypeOps  =
struct
  datatype t = NUM | STR

  fun arity NUM = []
    | arity STR = []

  fun equal (x : t, y : t) = x = y

  fun toString NUM = "num"
    | toString STR = "str"
end
```

As with terms, we can now use this structure to get a structure representing types, in `type.sml`:

```
structure Type = ABT_Util(Abt(TypeOps))
```

To implement the statics, we must write a typechecker. We can model a typechecker using the `TYPECHECKER` signature:

```
signature TYPECHECKER =
sig
  exception TypeError

  type context = Type.t Context.map

  val checktype : context -> Term.t -> Type.t
end
```

Here, a context is a map from variables to types (see `context.sml`).

Task 8 (10%). Implement the statics of $\mathcal{L}\{\texttt{num str}\}$ in the file `typechecker.sml`.

```
structure TypeChecker : TYPECHECKER =
struct
  exception TypeError
  exception NotImplemented

  type context = Type.t Context.map

  (* ... your solution goes here ... *)
   fun checktype ctx t  = raise NotImplemented
end
```

2.3 Unchecked Dynamics

Now that we have a statics, we can implement a simpler dynamics that does away with unnecessary error checking. In fact, in this case, we can get rid of all the error rules (of the form e err), though this is not always the case for every language (see discussion in Ch. 6 of PFPL).

Task 9 (10%). Implement unchecked dynamics for $\mathcal{L}\{\texttt{num str}\}$ in `uncheckeddynamics.sml`. Your abstract type `d` should be simpler than the one you used for the checked dynamics above.

2.4 Putting It Together

You can compile your files using `CM.make "sources.cm"`. We have provided three (!) ways to test the final implementation: an interpreter, a test harness and a reference implementation. All of these are based on a parser we provide for you (see examples below).

2.4.1 Interpreter

To run the interpreter, execute `TopLevel.repl();`. This will provide a command-line interpreter that will provide two basic commands, `step` and `eval`. Each of these commands take an additional argument *mode* which is either `checked` or `unchecked`, corresponding to the two kinds of dynamics you have implemented. We provide below the grammar that the interpreter accepts, as well as a sample session of the interpreter. Be aware of whether you are at the ML prompt '`-`', or the prompt for our language '`->`'. Ctrl-C will exit the interpreter.

```
command ::=
  step checked <exp>;
| step unchecked <exp>;
| step;
| eval checked <exp>;
| eval;

(* Whitespace is required even for +, *, and ^.
 * Infix operators associate left.
 * The precedence order is ^, +, *, len (so len binds tightest).
 *)
exp ::=
  <exp> + <exp>
| <exp> * <exp>
| <exp> ^ <exp>
| len <exp>
| <number>
| <string>
| <ident>
| let <decls> in <exp> end (* whitespace is required *)
| (<exp>)

decls ::=
  <decl>
| <decl> <decls>
```

```
decl ::= val <ident> = <exp>

number ::= (* one or more digits from 0 to 9 *)
string ::= (* letters, digits, spaces, and tabs enclosed in quotes (no newlines) *)
ident ::= (* a letter followed by any number of letters and numbers *)
```

Example interpreter session:

```
- TopLevel.repl();
->step checked 5+8;
 --> 13()

->step unchecked 5+8;
Statics : exp has type : num()
 --> 13()

->step checked let val x = "t" in x ^ "p" end;
 --> cat("t"(), "p"())

->step;
 --> "tp"()

->eval checked len("abc");
 3() VAL

->step checked 5 + 6 + 7 + 8;
 --> plus(plus(11(), 7()), 8())

->eval;
 26() VAL

->step checked 5 + "six";
 ERR

->step unchecked 5 + "six";
TypeChecker error!
```

2.4.2 Test Harness

Another way to test your code is by `TestHarness.runalltests(v);` where `v` is a `bool` indicating whether you want verbose output or not. This is mostly just a framework set up for you, in `tests.sml`, with a few simple test cases. You are responsible for handing in a working solution. Although not sufficient, this means handing in a well-tested implementation. You need to come up with test cases to exercise your code. In order to generate a comprehensive suite of tests, you are encouraged to share test cases with your classmates.

2.4.3 Reference Implementation

Finally, we have included the solution to this assignment as a binary heap image, `ref_impl`. You can load it into SML by passing in the `@SMLload=ref_impl` flag. Your solution should behave just like ours (if you find a bug in our implementation, extra credit to you!)

A Dynamics of $\mathcal{L}\{\texttt{num str}\}$

$\boxed{e \text{ val}}$

$$\frac{}{\texttt{num}[n] \text{ val}} \qquad \frac{}{\texttt{str}[s] \text{ val}}$$

$\boxed{e \mapsto e'}$

$$\frac{}{\texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) \mapsto \texttt{num}[n_1 + n_2]} \qquad \frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{plus}(\texttt{num}[n_1];e_2) \mapsto \texttt{plus}(\texttt{num}[n_1];e_2')} \qquad \frac{}{\texttt{times}(\texttt{num}[n_1];\texttt{num}[n_2]) \mapsto \texttt{num}[n_1 * n_2]}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{times}(e_1;e_2) \mapsto \texttt{times}(e_1';e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{times}(\texttt{num}[n_1];e_2) \mapsto \texttt{times}(\texttt{num}[n_1];e_2')}$$

$$\frac{}{\texttt{cat}(\texttt{str}[s_1];\texttt{str}[s_2]) \mapsto \texttt{str}[s_1 \hat{} s_2]} \qquad \frac{e_1 \mapsto e_1'}{\texttt{cat}(e_1;e_2) \mapsto \texttt{cat}(e_1';e_2)}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{cat}(\texttt{str}[s_1];e_2) \mapsto \texttt{cat}(\texttt{str}[s_1];e_2')} \qquad \frac{}{\texttt{len}(\texttt{str}[s]) \mapsto \texttt{num}[|s|]} \qquad \frac{e \mapsto e'}{\texttt{len}(e) \mapsto \texttt{len}(e')}$$

$$\frac{e_1 \text{ val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2} \qquad \frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x.e_2) \mapsto \texttt{let}(e_1';x.e_2)}$$

$\boxed{e \text{ err}}$

$$\frac{}{\texttt{plus}(\texttt{str}[s];e_2) \text{ err}} \qquad \frac{}{\texttt{plus}(\texttt{num}[n];\texttt{str}[s]) \text{ err}} \qquad \frac{e_1 \text{ err}}{\texttt{plus}(e_1;e_2) \text{ err}} \qquad \frac{e_2 \text{ err}}{\texttt{plus}(\texttt{num}[n];e_2) \text{ err}}$$

$$\frac{}{\texttt{times}(\texttt{str}[s];e_2) \text{ err}} \qquad \frac{}{\texttt{times}(\texttt{num}[n];\texttt{str}[s]) \text{ err}} \qquad \frac{e_1 \text{ err}}{\texttt{times}(e_1;e_2) \text{ err}} \qquad \frac{e_2 \text{ err}}{\texttt{times}(\texttt{num}[n];e_2) \text{ err}}$$

$$\frac{}{\texttt{cat}(\texttt{num}[n];e_2) \text{ err}} \qquad \frac{}{\texttt{cat}(\texttt{str}[s];\texttt{num}[n]) \text{ err}} \qquad \frac{e_1 \text{ err}}{\texttt{cat}(e_1;e_2) \text{ err}} \qquad \frac{e_2 \text{ err}}{\texttt{cat}(\texttt{str}[s];e_2) \text{ err}}$$

$$\frac{}{\texttt{len}(\texttt{num}[n]) \text{ err}} \qquad \frac{e \text{ err}}{\texttt{len}(e) \text{ err}} \qquad \frac{e_1 \text{ err}}{\texttt{let}(e_1;x.e_2) \text{ err}}$$

B Statics of $\mathcal{L}\{\texttt{num str}\}$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \qquad \frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \qquad \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \qquad \frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2}$$