

# Assignment #3: Type Inference

Roger Wolff for 15-312: Principles of Programming Languages

Out: Thursday, February 9th  
Due: Tuesday, February 21st 11:59pm

## 1 Introduction

For this assignment, we will be dealing with two statically-typed languages. The first is a language with functions, products, sums, and recursive types. We will call this language  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ . This language is presented in *PFPL* chapters 10-12,16. It is a fully annotated program. That is, looking at any particular piece of syntax will enable you to easily read off the type. This is the *internal* language, for which we will define a runtime semantics.

We will also define an external language  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ . This language will allow us to infer types (very much like ML does) for certain expressions without requiring explicit annotations.

We will define an elaboration layer that typechecks an expression in  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ , while at the same time translating it to an equivalent expression in  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ . This elaboration layer will also allow us to provide more primitive types, like `bool` and `nat`, without having to overly complicate the internal language.

### 1.1 Submission

To submit your solutions place your `assn3.pdf` and `*.sml *.lam` files in your handin directory:

`/afs/andrew.cmu.edu/course/15/312/handin/<yourandrewid>/assn3/`

## 2 Hello, Types!

We started the course by introducing the untyped (or *untyped*)  $\lambda$ -calculus. All expressions were well-formed, and all expressions could successfully be evaluated. By successful, we do not mean that it would produce the desired result, but only that it would never fault.

We then introduced another untyped language,  $\mathcal{L}\{\text{Dyn}\}$ . This dynamic language had *classes*, but not types. These classes (like `num` or `fun`) could enable us at runtime to determine whether or not our program made sense. For example, during evaluation it is possible to determine that a number was being applied to another expression. In that case, the program is blatantly wrong, so runtime errors were introduced to signal

the fault.

Oh, if only we could prevent these faulty programs from ever being run. We are now introducing *types* to do precisely that. The statics of the language are now non-trivial as they need to enforce that only functions may be applied to an argument, and if the function, say, is expecting a number, then the argument had better be a number.

Appendix A defines our internal language  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ .

### 3 Goodbye, Types!

One of the purported benefits of dynamically-typed languages is that it unburdens the programmer from having to annotate types all over the place. But doing away with annotations is not the same as doing away with types. We will introduce an external language  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  that does not require annotations. We will attempt to infer a valid typing for a program. In some cases, the compiler will not be able to infer all necessary types. In these cases, we allow for type *ascription*, in which the programmer writes down the type of expression. For example, the expression  $\lambda x.x$  can have infinitely many types – how is the compiler to choose? For this case, the programmer can ascribe the type  $\text{nat} \rightarrow \text{nat}$  to the expression in the form  $(\lambda x.x) : (\text{nat} \rightarrow \text{nat})$  which will allow the compiler to assert that that is a valid type for  $\lambda x.x$ .

We divide the syntax into two types of expressions, *synthesis* terms and *analysis* terms. Synthesis expressions can produce their own type. For example, zero can synthesize type  $\text{nat}$ . The type of analysis terms cannot be determined in isolation. Instead, they are checked for compatibility with a given type. So  $\lambda x.x$  can be analyzed successfully against  $\text{nat} \rightarrow \text{nat}$ , or  $\text{bool} \rightarrow \text{bool}$ , but not  $\text{nat} \rightarrow \text{bool}$ .

#### 3.1 Syntax

$\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  has the same types as  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ . The expressions are similar to that of the internal language, with the absence of most types.

Analysis expressions	$a ::= s$	synthesis terms
	$\langle \rangle$	null tuple
	$\langle a_1, a_2 \rangle$	binary product
	$\lambda x. a$	$\lambda$ -abstraction
	$l \cdot a$	left injection
	$r \cdot a$	right injection
	$\text{case } (s; x_1.a_1; x_2.a_2)$	case analysis
	$\text{fold}(a)$	constructor
	$\text{abort } a$	abort
Synthesis expressions	$s ::= x$	variables
	$s a$	application
	$s \cdot l$	left projection
	$s \cdot r$	right projection
	$\text{unfold}(s)$	destructor
	$a \upharpoonright \tau$	type ascription

### 3.2 Bidirectional Typechecking and Elaboration

We define two judgements in order to translate the external language to the internal language. The first,  $\Gamma \vdash e \Rightarrow e' : \tau$ , says that in context  $\Gamma$ , the external language expression  $e$  synthesizes a type  $\tau$ , and at the same time, elaborates this to an expression  $e'$  of the internal language. The second judgement,  $\Gamma \vdash e \Leftarrow e' : \tau$  states that under context  $\Gamma$ , we can analyze the external language expression  $e$  at type  $\tau$ , and elaborate this to an expression  $e'$  of the internal language. In each of the judgements,  $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ , for some  $n \geq 0$ .

These *simultaneous induction* definitions serve a dual purpose. First, they typecheck the external language. And second, they provide a translation to the internal language. The forms of expressions in both languages are identical (so far) with two exceptions. First is the noticeable lack of types. As we type the external language, we translate each implicitly typed expression to its corresponding explicitly typed expression.

Upon first reading of these rules it may be useful to ignore the translation part of the rule. That is,  $\Gamma \vdash e \Rightarrow e' : \tau$  can be read instead read as  $\Gamma \vdash e \Rightarrow \tau$  to mean that  $e$  can synthesize  $\tau$ . Similarly,  $\Gamma \vdash e \Leftarrow e' : \tau$  can be read as  $\Gamma \vdash e \Leftarrow \tau$ , that  $e$  can analyze  $\tau$ .

These rules define an external language in isolation from the internal one. If we wanted, we could define a dynamic semantics for this external language. We could then show (as we did in Homework #2) that computation of the translation to the internal language faithfully simulates the computation of the original expression in the external language. Since the languages are so similar, we will forgo this exercise, and instead not even define a dynamics to the external language, and simply provide a translation.

While *global* type inference would allow us to write code without any types, we do not have the necessary tools at our disposal to accomplish this. Instead, we use a simpler *local* type inference. With only local type inference at our disposal, some terms cannot be typed without help. To recover from the missing type information, we provide type ascription, in the form  $e \upharpoonright \tau$  to give expression  $e$  type  $\tau$ . In this way, we can now type the expression  $\lambda x. x \upharpoonright \text{nat} \rightarrow \text{nat}$  to be the identity function on natural numbers.

### 3.2.1 Analysis

$$\begin{array}{c}
\frac{\Gamma \vdash s \Rightarrow s' : \tau}{\Gamma \vdash s \Leftarrow s' : \tau} (\Leftarrow S) \quad \frac{}{\Gamma \vdash \langle \rangle \Leftarrow \langle \rangle : \text{unit}} (\Leftarrow \text{TRIV}) \quad \frac{\Gamma \vdash a_1 \Leftarrow a'_1 : \tau_1 \quad \Gamma \vdash a_2 \Leftarrow a'_2 : \tau_2}{\Gamma \vdash \langle a_1, a_2 \rangle \Leftarrow \langle a'_1, a'_2 \rangle : \tau_1 \times \tau_2} (\Leftarrow \text{PAIR}) \\
\\
\frac{\Gamma, x : \tau_1 \vdash a \Leftarrow a' : \tau_2}{\Gamma \vdash \lambda x. a \Leftarrow (\lambda x : \tau_1. a') : \tau_1 \rightarrow \tau_2} (\Leftarrow \lambda) \quad \frac{\Gamma \vdash a \Leftarrow a' : [\mu t. \tau / t] \tau}{\Gamma \vdash \text{fold}(a) \Leftarrow \text{fold}[t. \tau](a') : \mu t. \tau} (\Leftarrow \text{FOLD}) \\
\\
\frac{\Gamma \vdash a \Leftarrow a' : \tau_1}{\Gamma \vdash l \cdot a \Leftarrow \text{inl}[\tau_1 + \tau_2](a') : \tau_1 + \tau_2} (\Leftarrow \text{INJL}) \quad \frac{\Gamma \vdash a \Leftarrow a' : \tau_2}{\Gamma \vdash r \cdot a \Leftarrow \text{inr}[\tau_1 + \tau_2](a') : \tau_1 + \tau_2} (\Leftarrow \text{INJR}) \\
\\
(\Leftarrow \text{CASE}) \frac{\Gamma \vdash s \Rightarrow s' : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash a_1 \Leftarrow a'_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash a_2 \Leftarrow a'_2 : \tau}{\Gamma \vdash \text{case}(s; x_1. a_1; x_2. a_2) \Leftarrow \text{case}(s'; x_1. a'_1; x_2. a'_2) : \tau} \quad \frac{\Gamma \vdash a \Leftarrow a' : \text{void}}{\Gamma \vdash \text{abort } a \Leftarrow \text{abort}[\tau] a' : \tau} (\Leftarrow \text{ABORT})
\end{array}$$

Analysis rules can be thought of as requiring the type of an expression as input. The term  $\lambda x. a$  can be typed  $\tau_1 \rightarrow \tau_2$ , if, assuming  $x : \tau_1$  the body  $a$  can be typed at  $\tau_2$  ( $\Leftarrow \lambda$ ). The one interesting rule here is ( $\Leftarrow S$ ), which states that any synthesis term is an analysis term. If  $s$  can synthesize  $\tau$ , then it can also be analyzed against it ( $\Leftarrow S$ ).

### 3.2.2 Synthesis

$$\begin{array}{c}
\frac{}{\Gamma, x : \tau \vdash x \Rightarrow x : \tau} (\Rightarrow \text{VAR}) \quad \frac{\Gamma \vdash s \Rightarrow s' : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash a \Leftarrow a' : \tau_1}{\Gamma \vdash s a \Rightarrow s' a' : \tau_2} (\Rightarrow \text{APP}) \\
\\
\frac{\Gamma \vdash s \Rightarrow s' : \tau_1 \times \tau_2}{\Gamma \vdash s \cdot l \Rightarrow s' \cdot l : \tau_1} (\Rightarrow \text{PROJ}_l) \quad \frac{\Gamma \vdash s \Rightarrow s' : \tau_1 \times \tau_2}{\Gamma \vdash s \cdot r \Rightarrow s' \cdot r : \tau_2} (\Rightarrow \text{PROJ}_r) \\
\\
\frac{\Gamma \vdash s \Rightarrow s' : \mu t. \tau}{\Gamma \vdash \text{unfold}(s) \Rightarrow \text{unfold}(s') : [\mu t. \tau / t] \tau} (\Rightarrow \text{UNFOLD}) \quad \frac{\Gamma \vdash a \Leftarrow a' : \tau}{\Gamma \vdash a \upharpoonright \tau \Rightarrow a' : \tau} (\Rightarrow \upharpoonright)
\end{array}$$

Unlike analysis terms, the type for synthesis terms could be thought of as an output from the typing rules. In order to synthesize the type for a variable,  $x$ , we simply read off its type from the context ( $\Rightarrow \text{VAR}$ ). To synthesize a type for an application, we first synthesize the type for the function. We can then use the more general form and analyze its argument against the prescribed type. The term  $a \upharpoonright \tau$  is given type  $\tau$  if  $a$  can be analyzed against it ( $\Rightarrow \upharpoonright$ ).

## 3.3 Derived Forms

Elaborating terms from an external language to an internal language provides an additional benefit. We can add *syntactic sugar*, new terms to the external language without having to make the internal language more complicated.

Here, we add several new terms to allow us to write more sensible looking code. Numbers are elaborated into their unary equivalent. The let expression, let  $x = e_1$  in  $e_2$ , as you would see in languages such as ML, binds  $e_1$  to  $x$  in  $e_2$ . Booleans are defined with a sum type, and numbers are defined by a recursive type.

Types	$t ::=$	nat bool	natural numbers boolean
Analysis expressions	$a ::=$	fix $x.a$ if( $a; a_1; a_2$ ) ifz( $a; a_1; x.a_2$ )	general recursion boolean conditional num conditional
Synthesis expressions	$s ::=$	$\bar{n}$ zero succ( $a$ ) tt ff let $x = s_1$ in $s_2$	$n \in \mathbb{N}$ zero successor true false local variable

We can add types to our external language, but that would require us to also provide translations of the types as well as expressions. Since this adds unnecessary complications, we will just take the additional types as the following aliases:

$$\begin{aligned} \text{nat} &= \mu t. \text{unit} + t \\ \text{bool} &= \text{unit} + \text{unit} \end{aligned}$$

$$\frac{\Gamma \vdash \text{zero} \Leftarrow z : \text{nat}}{\Gamma \vdash \bar{0} \Rightarrow z : \text{nat}} (\Rightarrow \mathbf{N}_0) \qquad \frac{\Gamma \vdash \text{succ}(\bar{n}) \Leftarrow n' : \text{nat}}{\Gamma \vdash \bar{n} + \bar{1} \Rightarrow n' : \text{nat}} (\Rightarrow \mathbf{N}_1)$$

$$(\Rightarrow \mathbf{ZERO}) \frac{}{\Gamma \vdash \text{zero} \Rightarrow \text{fold}[t.\text{unit} + t](\text{inl}[\text{unit} + \text{nat}]()) : \text{nat}} \qquad \frac{\Gamma \vdash a \Leftarrow a' : \text{nat}}{\Gamma \vdash \text{succ}(a) \Rightarrow \text{fold}[t.\text{unit} + t](\text{inr}[\text{unit} + \text{nat}](a')) : \text{nat}} (\Rightarrow \mathbf{SUCC})$$

$$\frac{}{\Gamma \vdash \text{tt} \Rightarrow \text{inl}[\text{unit} + \text{unit}]() : \text{bool}} (\Rightarrow \mathbf{TT}) \qquad \frac{}{\Gamma \vdash \text{ff} \Rightarrow \text{inr}[\text{unit} + \text{unit}]() : \text{bool}} (\Rightarrow \mathbf{FF})$$

$$\frac{\Gamma, x : \tau \vdash a \Leftarrow a' : \tau}{\Gamma \vdash \text{fix } x.a \Leftarrow \text{unroll}(\text{self}[\tau](y. [\text{unroll}(y)/x]a')) : \tau} (\Leftarrow \mathbf{FIX})$$

The rule ( $\Leftarrow$ Fix) requires the following definitions:

$$\begin{aligned} \text{self}(\tau) &= \mu t. t \rightarrow \tau \\ \text{self}[\tau](x.e) &= \text{fold}[t.t \rightarrow \tau](\lambda x : \text{self}(\tau).e) \\ \text{unroll}(e) &= \text{unfold}(e)(e) \end{aligned}$$

**Task 3.1** [3 points]:

Give the elaboration rules for let, if, ifz.

### 3.4 Evaluation Strategy

The languages presented in class so far have been call by value. Subexpressions are reduced eagerly. There are several other common evaluation strategies, some of which we will discuss later in class. The language  $\mathcal{L}_i\{\rightarrow \times + \mu\}$  uses a call by name strategy. This is a *lazy* strategy in that expressions are not evaluated until they are needed by the *elimination* rules. For example, consider the expression  $\text{ifz}(e_1; e_2; x.e_3)$ . In order to make progress, we don't need to know what number  $e_1$  evaluates to, only whether it evaluates to zero or not. Therefore, we always treat  $\text{succ}(e'_1)$  as a value, regardless of the form of  $e'_1$ . We treat applications, pairs, and injections similarly.

**Task 3.2** [5 points]:

Due to this laziness, we can no longer compare two numbers simply by testing for  $\alpha$ -equivalence. Write an expression of type  $\text{nat} \rightarrow \text{nat} \rightarrow \text{bool}$  to compare two numbers for equality.

## 4 Safety

The safety of our  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  will be proven by three main theorems. First, we will show that there is *coherence* between the external and internal languages. If an expression  $e$  of the external language is typed at  $\tau$  then the elaboration of  $e$  produces an expression  $e'$  of the internal language that also types at  $\tau$ . We then prove the standard progress and preservation theorems of  $\mathcal{L}_i\{\rightarrow \times + \mu\}$  showing that evaluating our expression will never fail, and that, if it terminates, will result in a value of type  $\tau$ .

Before we get to those, here are a couple of lemmas you may find useful. Both the substitution and weakening lemmas are standard and will hold for all languages we study in class<sup>1</sup>.

**Lemma 1** (Substitution). *If  $\Gamma, x : \tau_2 \vdash e_1 : \tau_1$  and  $\Gamma \vdash e_2 : \tau_2$  then  $\Gamma \vdash [e_2/x]e_1 : \tau_1$ .*

*Proof.* We can prove Lemma 1 by induction on the derivation of  $\Gamma, x : \tau_2 \vdash e_1 : \tau_1$  and the definition of substitution, which is not explicitly defined for this language, but is as you would expect.  $\square$

**Lemma 2** (Weakening). *If  $\Gamma \vdash e : \tau$ , then  $\Gamma, x : \tau' \vdash e : \tau$ , provided that  $x \notin \text{domain}(\Gamma)$ .*

*Proof.* Induction on derivation of  $\Gamma \vdash e : \tau$ .  $\square$

**Theorem 1** (Coherence). *If  $\Gamma \vdash s \Rightarrow s' : \tau$ , then  $\Gamma \vdash s' : \tau$ . If  $\Gamma \vdash a \Leftarrow a' : \tau$ , then  $\Gamma \vdash a' : \tau$ .*

The coherence theorem relates the elaboration of the external to internal languages. Although the terms and judgements look similar, keep in mind that they are for entirely different languages. Expressions  $s, a$  are of the  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ , and  $s', a'$  are of the  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ . Types and contexts are the same between languages.

**Task 4.1** [10 points]:

Theorem 1 can be proven by simultaneous induction on the derivations of  $\Gamma \vdash s \Rightarrow s' : \tau$  and  $\Gamma \vdash a \Leftarrow a' : \tau$ . Show the cases for ( $\Rightarrow$  ASC), ( $\Leftarrow$  S), and ( $\Leftarrow$  Fix).

---

<sup>1</sup>Not all logics have the property of weakening. When weakening is removed, we are dealing with a *linear* logic, which is a type of *substructural* logic

We are now finished dealing with the external language. The following theorems only apply to  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ .

A progress theorem states that the behavior of a program is well-defined. Either an expression can make take a step (make progress), or else it must be an irreducible expression (a value).

**Theorem 2 (Progress).** *If  $e : \tau$ <sup>2</sup>, then either  $e \mapsto e'$  for some  $e'$ , or  $e \text{ val}$ .*

In order to prove the progress theorem, we need to introduce a canonical forms lemma. The canonical forms lemma states what can be determined by knowing the type of a value.

**Lemma 3 (Canonical forms).** *If  $e \text{ val}$  and  $e : \tau$ ,*

1. *If  $\tau = \text{nat}$ , then  $e = \text{zero}$  or  $e = \text{succ}(e')$  for some  $e'$ , such that  $e' : \text{nat}$ .*
2. *If  $\tau = \tau_1 \rightarrow \tau_2$ , then  $e = \lambda x : \tau_1. e'$  for some  $x, e'$ , such that  $x : \tau_1 \vdash e' : \tau_2$ .*
3. *If  $\tau = \text{unit}$ , then  $e = \langle \rangle$ .*
4. *If  $\tau = \tau_1 \times \tau_2$ , then  $e = \langle e_1, e_2 \rangle$ , for some  $e_1, e_2$ , such that  $e_1 : \tau_1$  and  $e_2 : \tau_2$ .*
5. *If  $\tau = \tau_1 + \tau_2$ , then either  $e = \text{inl}[\tau_1 + \tau_2](e_1)$  for some  $e_1$ , and  $e_1 : \tau_1$  or  $e = \text{inr}[\tau_1 + \tau_2](e_2)$  for some  $e_2$ , and  $e_2 : \tau_2$ .*

*Proof.* We can prove Lemma 3 by induction on the typing rules with the assumption that  $e \text{ val}$ . □

**Task 4.2** [5 points]:

Theorem 2 can now be proven by induction on the derivation of  $e : \tau$ . Prove the progress theorem for cases (APP), (CASE).

The other half of the safety proof states is preservation. While progress states that an expression is either a value or can take a step, the preservation theorem says that if an expression can take a step, then its type does not change. In practical terms, the progress theorem ensures that the program is well-defined (no blue screens, core dumps, method not found, ...), preservation ensures that your program will compute what it is supposed to compute (with respect to types). If you are computing a `nat`, your program might very well result in an incorrect answer, but at least it will be a `nat`.

**Theorem 3 (Preservation).** *If  $e : \tau$  and  $e \mapsto e'$ , then  $e' : \tau$ .*

In order to prove preservation, we will need to make use of an inversion lemma.

**Lemma 4 (Inversion).** *If  $\Gamma \vdash e : \tau$  then*

1. *If  $e = x$ , then  $x : \tau \in \Gamma$ .*
2. *If  $e = \lambda x : \tau_1. e'$ , then  $\tau = \tau_1 \rightarrow \tau_2$  for some  $\tau_2$ , and  $\Gamma, x : \tau_1 \vdash e' : \tau_2$ .*
3. ...

---

<sup>2</sup>We are abbreviating  $\cdot \vdash e : \tau$  as  $e : \tau$ . That is to say  $e$  has type  $\tau$  with the empty context, a closed term with no free variables.

In effect, the inversion lemma lets us read the typing rules in reverse. This is trivially proven by induction, as our language has exactly one rule per form. We have left out many of the cases of the inversion lemma for brevity.

**Task 4.3** [2 points]:

What does the inversion lemma state for the case where  $e = \langle e_1, e_2 \rangle$ ?

**Task 4.4** [6 points]:

Theorem 3 can be proven by induction on the derivation of  $e \mapsto e'$ . Prove the preservation theorem for cases  $(\mapsto \text{PROJ}_{11})$  and  $(\mapsto \text{CASE}_2)$ .

## 5 Implementation

As usual, first we design and study the language. And then we get to implement it and see the formalism come to life.

### 5.1 Abstract Binding Trees

For Homework #2, you implemented three different languages. Each language necessarily had their own abstract syntax, but for each language, a lot of duplicated code was necessary to perform what should be language-agnostic manipulations. Take substitution, for example. In order to substitute  $e_1$  for  $x$  in  $e_2$ , we only need to be able to traverse the abstract binding tree (ABT) and be able to rebuild it with  $e_1$  in place of each occurrence of  $x$ .

Toward this purpose, we have built a language-agnostic ABT. The functor `ABT_Util` requires an `OPERATOR` that will represent language features. It will then act in a very similar manner as your ABTs in homework. In order to construct an ABT, an `OPERATOR` is provided. The key to the `OPERATOR` signature is the function `arity`<sup>3</sup>, which takes a language-defined type `t`, and returns an array of integers,  $[b_1, b_2, \dots, b_n]$ , where  $n$  is the number of arguments that the operator takes, and  $b_i$  is the number of binders of the  $i^{\text{th}}$  argument. For example, the expression `ifz(e0; e1; x.e2)` would have arity  $[0, 0, 1]$ , since there are 3 arguments and one binder  $x$  in the third argument position.

There are three ABTs that you will need to use in this assignment. One is for the external language, `TermExt`, using `TermExtOps` operators. One is for the internal language `TermInt` using `TermIntOps`. And one is for the types `Type`, with operators `TypeOps`. The same types will be used for both the external and internal languages, so there is no need to translate from one to the other. All three ABTs are given to you.

The code should look very familiar. To construct an ABT, `into` is called with either a variable, a binder coupled with a term over which the binder ranges, or an operator and a list of arguments. One difference between the ABTs of last week's assignment is in the signature of the substitution function. `subst e1 x e2` performs the substitution,  $[e_1/x]e_2$ .

---

<sup>3</sup>See chapter 1 of *PFPL*

## 5.2 Elaborator

The elaborator is the compiler for our language. It has the following signature for you to implement:

```
signature ELABORATOR =
sig
  exception Type_Error of string

  type context = Type.t Context.dict

  val analyze : context -> TermExt.t -> Type.t -> TermInt.t
  val synthesize : context -> TermExt.t -> TermInt.t * Type.t
end
```

The functions `analyze` and `synthesize` are described in Section 3. The `context` argument will adhere to the following signature:

```
signature DICT =
sig

  type key
  type 'a dict

  ...
  val insert : 'a dict -> key -> 'a -> 'a dict
  val find : 'a dict -> key -> 'a option
  val lookup : 'a dict -> key -> 'a
  ...
end
```

The complete signature can be found in `cmlib/dict.sig` if any additional functionality is needed. The keys will be of type `Variable.t`, and the values of of type `Type.t`, the ABT of the type of the variable.

### Task 5.1 [10 points]:

Implement the elaborator. The function `analyze` takes an expression from the  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  and a type as input and produces an expression of the  $\mathcal{L}_i\{\rightarrow \times + \mu\}$  of the provided type. If this is not possible, a `Type_Error` should be raised. The function `synthesize` does the analogous operation, and provides both the translated term and the synthesized type as the result.

## 5.3 Typechecker

The typechecker is not strictly necessary, because the elaboration performs type checking as well. If your program elaborates successfully, then the safety theorems will hold, and if you are expected a computation of type `nat`, say, then a `nat` is what you will get. But the type checker is not necessary only if you can really trust your translation code. For the fallibility in all of us, we can use the typechecker as a verification tool that the translation is done successfully. It doesn't assure that anything is really right, but it at least forces the more complicated elaboration code to agree with the typechecker's result checking simpler code for a simpler language.

```
signature TYPECHECKER =
sig
  exception Type_Error of string

  type context = Type.t Context.dict
```

```

    val check : context -> TermInt.t -> Type.t
end

```

The `context` is the same as with the elaborator. According to Theorem 1, if `Elaborator.analyze ctx a t` returns `a'`, then `TypeChecker.check ctx a'` must evaluate to `t`. Similarly, if `Elaborator.synthesize ctx a t` return `(a', t)` then `TypeChecker.check ctx a'` must evaluate to `t` as well. The starter code will enforce this.

**Task 5.2** [5 points]:

Implement the `TypeChecker` according to the rules in Appendix A.2<sup>4</sup>.

## 5.4 Evaluator

The evaluator is the interpreter for the  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ .

```

signature EVALUATOR =
sig
  exception Runtime_Error of string

  val value : TermInt.t -> bool
  val step : TermInt.t -> TermInt.t
end

```

**Task 5.3** [5 points]:

Implement the `Evaluator`. This should be a straight-forward translation of the value and transition judgements of Appendix A.3.

## 5.5 $\mathcal{L}(Dyn)$ as an Embedded Languages

Now that we've implemented  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ , let's write some code!<sup>5</sup>

Recall the dynamic language presented in class and implemented as part of Homework #2. We will write an embedded language to run  $\mathcal{L}(Dyn)$ . That is, we will not create a new parser and a new interpreter. Instead, we will write  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  terms that faithfully emulate  $\mathcal{L}(Dyn)$ .

All code should be placed in a file named `dyn.lam` and submitted with the rest of the assignment.

The code we write will not look exactly like the syntax you might expect (or want), but it is pretty enlightening to do so. First, we must embed the type `dyn` into  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ , as  $\mu t.\text{nat} + (t \rightarrow t)$ . This is a recursive type that is either a `nat`, or else it is a function from `dyn` to `dyn`<sup>6</sup>.

**Task 5.4** [3 points]:

---

<sup>4</sup>The rules to check for valid types in Appendix A.2.1 are included for your edification only. All types will be verified by the parser. Also note that the statics for both  $\mathcal{L}_i\{\rightarrow \times + \mu\}$  and  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  implicitly assume that all types mentioned are well-formed according to the judgement  $\tau$  type

<sup>5</sup>The static syntax has changed significantly from the last assignment. See `readme.lam` for sample code and documentation.

<sup>6</sup>For simplicity, we are forgoing the class of pairs as presented originally in the language. Doing so would just require a four-way sum to account for `nil` and `cons`

Write constructors `dnum` and `dfun` to create data of type `dyn`, when given the underlying `nat` or `dyn → dyn`, respectively.

Before we can continue, we need to address the possibility of errors. Since  $\mathcal{L}(Dyn)$  may error (`dnum (0)` `dnum (0)`, say) we need to “enhance” the capabilities of our statically-typed language which cannot fail, to a language that can. To this regard, the expression `error` of  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  should be translated by the elaborator to  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ , and an exception should be raised if the expression ever tries to be evaluated. The formalism would still be propagating errors, but the implementation can accomplish this by exceptions. Once `error` is added to the language, the safety theorems need to be restated to deal with the possibility of errors.

**Task 5.5** [2 points]:

Restate the theorems that need to be changed.

We now have to deal with a couple of technical details. The first is that typing judgement for application,  $d_0 : \text{dyn}, d_1 : \text{dyn} \vdash d_0 d_1 : \text{dyn}$ , is not valid in  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ . To get around this we will need to implement `dcall`: `dyn → dyn → dyn`. Function applications will be written as `dcall d0 d1`.

The second technicality comes from the impossibility of writing the raw expression with a binder that is required for `ifz(d0; d1; x.d2)`. Instead, we will implement `difz`: `dyn → dyn → (dyn → dyn)`. If `d0` is nonzero, then the function `d2` should be called with an argument of the predecessor of `d0`. On one hand, we may think that we are getting around our embedded language because a term of type `dyn → dyn` is not representable in  $\mathcal{L}(Dyn)$ . But on the other hand, we can think of this as a direct translation `ifz(d0; d1; x.d2)` to `difz d0 d1 (fn x => d2)`.

**Task 5.6** [5 points]:

Add `dzero`, `dsucc`, `difz`, `dcall` expressions.

The last technicality is that due to the limitations of our typechecking algorithm, it may be necessary to annotate expressions with their type (it’s always `dyn` of course).

## 5.6 Testing

Finally, we can write some interesting code in either the embedded  $\mathcal{L}(Dyn)$ , or in our full language  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$ .

**Task 5.7** [10 points]:

Write a (very) small program in each language, along with other test cases to validate your code.

In file `debug.sml`, there is some limited diagnostic information. Turning on the switch `Debug.typecheckEveryStep` will typecheck the program at every step of evaluation.

## 5.7 Submission

You should turn in files `evaluator.sml`, `typechecker.sml`, `elaborator.sml` for the implementation of  $\mathcal{L}_e\{\rightarrow \times + \mu \text{ nat bool}\}$  and  $\mathcal{L}_i\{\rightarrow \times + \mu\}$ , `dyn.lam` for the implementation of  $\mathcal{L}(\text{Dyn})$  along with code in the embedded language, and you should turn in `tests.lam` for any other miscellaneous tests.

## A Internal Language, $\mathcal{L}_i\{\rightarrow \times + \mu\}$

### A.1 Syntax

Types	$\tau ::=$	$t$	type variable
		$\mu t. \tau$	recursive type
		$\tau_1 \rightarrow \tau_2$	partial function
		$\text{unit}$	nullary product
		$\tau_1 \times \tau_2$	binary product
		$\text{void}$	nullary sum
		$\tau_1 + \tau_2$	binary sum
Expression	$e ::=$	$x$	variable
		$\lambda x : \tau. e$	$\lambda$ -abstraction
		$e_1 e_2$	application
		$\langle \rangle$	null tuple
		$\langle e_1, e_2 \rangle$	binary product
		$e \cdot l$	left projection
		$e \cdot r$	right projection
		$\text{abort}[\tau]e$	abort
		$\text{inl}[\tau_1 + \tau_2](e)$	left injection
		$\text{inr}[\tau_1 + \tau_2](e)$	right injection
		$\text{case}(e_0; x_1.e_1; x_2.e_2)$	case analysis
		$\text{fold}[t.\tau](e)$	constructor
		$\text{unfold}(e)$	destructor

### A.2 Statics

#### A.2.1 Type rules

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \times \tau_2 \text{ type}} \text{ (TPAIR)} \qquad \frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}} \text{ (T}\lambda\text{)}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 + \tau_2 \text{ type}} \text{ (TSUM)} \qquad \frac{}{\Delta \vdash \text{unit type}} \text{ (TUNIT)} \qquad \frac{}{\Delta \vdash \text{void type}} \text{ (TVOID)}$$

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}} \text{ (TVAR)} \qquad \frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \mu t. \tau \text{ type}} \text{ (TREC)}$$

#### A.2.2 Expression rules

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \text{ (VAR)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. e) : \tau_1 \rightarrow \tau_2} (\lambda) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1} \text{ (APP)}$$

$$\frac{}{\Gamma \vdash \langle \rangle : \text{unit}} \text{ (UNIT)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{ (PROD)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot l : \tau_1} \text{ (PROJ}_l\text{)} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot r : \tau_2} \text{ (PROJ}_r\text{)} \quad \frac{\Gamma \vdash e : \text{void}}{\Gamma \vdash \text{abort}[\tau]e : \tau} \text{ (ABORT)}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}[\tau_1 + \tau_2](e) : \tau_1 + \tau_2} \text{ (INJ}_l\text{)} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}[\tau_1 + \tau_2](e) : \tau_1 + \tau_2} \text{ (INJ}_r\text{)}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}(e_0; x_1.e_1; x_2.e_2) : \tau} \text{ (CASE)}$$

$$\frac{\Gamma \vdash e : [\mu t. \tau / t] \tau}{\Gamma \vdash \text{fold}[t. \tau](e) : \mu t. \tau} \text{ (FOLD)} \quad \frac{\Gamma \vdash e : \mu t. \tau}{\Gamma \vdash \text{unfold}(e) : [\mu t. \tau / t] \tau} \text{ (UNFOLD)}$$

## A.3 Dynamics

### A.3.1 Values

$$\frac{}{\lambda x : T. e \text{ val}} \text{ (VAL-}\lambda\text{)} \quad \frac{}{\langle \rangle \text{ val}} \text{ (V}\langle \rangle_1\text{)} \quad \frac{}{\langle e_1, e_2 \rangle \text{ val}} \text{ (V}\langle \rangle_2\text{)}$$

$$\frac{}{\text{inl}[\tau_1 + \tau_2](e) \text{ val}} \text{ (VINJ}_l\text{)} \quad \frac{}{\text{inr}[\tau_1 + \tau_2](e) \text{ val}} \text{ (VINJ}_r\text{)} \quad \frac{}{\text{fold}[t. \tau](e) \text{ val}} \text{ (V}\mu\text{)}$$

### A.3.2 Transitions

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{ (}\mapsto\text{APP)} \quad \frac{}{(\lambda x : \tau. e) e_2 \mapsto [e_2/x]e} \text{ (}\mapsto\beta\text{)} \quad \frac{e \mapsto e'}{e \cdot l \mapsto e' \cdot l} \text{ (}\mapsto\text{PROJ}_{l1}\text{)}$$

$$\frac{}{\langle e_1, e_2 \rangle \cdot l \mapsto e_1} \text{ (}\mapsto\text{PROJ}_{l2}\text{)} \quad \frac{}{\langle e_1, e_2 \rangle \cdot r \mapsto e_2} \text{ (}\mapsto\text{PROJ}_{r2}\text{)} \quad \frac{e \mapsto e'}{e \cdot r \mapsto e' \cdot r} \text{ (}\mapsto\text{PROJ}_{r1}\text{)}$$

$$\frac{e_0 \mapsto e'_0}{\text{case}(e_0; x_1.e_1; x_2.e_2) \mapsto \text{case}(e'_0; x_1.e_1; x_2.e_2)} \text{ (}\mapsto\text{CASE}_1\text{)}$$

$$\frac{}{\text{case}(\text{inl}[\tau_1 + \tau_2](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1} \text{ (}\mapsto\text{CASE}_2\text{)}$$

$$\frac{}{\text{case}(\text{inr}[\tau_1 + \tau_2](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2} \text{ (}\mapsto\text{CASE}_3\text{)}$$

$$\frac{e \mapsto e'}{\text{unfold}(e) \mapsto \text{unfold}(e')} \text{ (}\mapsto\text{UNFOLD}_1\text{)}$$

$$\frac{}{\text{unfold}(\text{fold}[t. \tau](e)) \mapsto e} \text{ (}\mapsto\text{UNFOLD}_2\text{)}$$