CS251 Programming Languages **Handout # 33**
Prof. Lyn Turbak **April 18, 2001**
Wellesley College

Type Reconstruction

Note: The presentation of the unification and type reconstruction algorithms in this handout is simpler than the presentation in class and supersedes the class presentation. In particular, important changes include:

- The unification algorithm in `Subst.sml` has been rewritten so that the following functions presented in class are no longer necessary: `unifyWRT`, `unifyWRTList`, `unifyWRTVar`.
- In `Subst.sml`, what was called `subst`X in class has been renamed to `sub`X, where X ranges over `Ty`, `Tys`, `Exp`, `Exps`. The signatures of all these functions have changed so that they take their two arguments in curried fashion rather than as a tuple; this makes it easier to map them when they are applied to one argument.
- In `Subst.sml`, `occursIn` no longer does an explicit tree walk, but just does a membership test on the result of `Type.TVs`.
- In `Recon.sml`, in the triples returned by `reconExp`, it is no longer required that the first (expression) and second (type) components of the triple be canonical w.r.t. the third (substitution) component. The substitution accumulated during a program is applied only to only one expression (the program body) as part of returning an explicitly typed program.

1 Introduction

In the typed toy languages that we've studied so far (HOFLEMT and HOFLEPT), it is necessary to specify explicit type information in certain situations. For example, in HOFLEMT:

- All formal variables declared by `abs` must be accompanied by their types.
- All names declared by `bindrec` must have explicit types.
- An application of the primitive operator for creating an empty list must indicate the component type of the empty list.

In addition, the HOFLEPT language requires special constructs for introducing (`pabs`) and eliminating (`papp`) universally quantified (`forall`) types.

What determines the placement of explicit type information in a language? That is, why is it that some type information must be provided while other type information can be elided? The answer to this question lies in the structure of the type checker. As noted earlier, a simple type checker has the structure of an evaluator. Consider type checking an abstraction. When entering an abstraction, the type checker has no information about the types of the formal parameters; these must be provided explicitly. However, once the types of the formals is known, it is easy for the type checker to determine the type of the body, so this information need not be declared.

Could a more sophisticated type checker do its job with even less explicit information? Certainly, programmers can reason proficiently about type information in many programs where there are no explicit types at all. Such reasoning is important because understanding the type of an expression, especially one that denotes a procedure, is often a major step in figuring out what purpose the expression serves in the program. As an example of this kind of type reasoning, consider the following HOFL expression:

```
(abs (f g x y)
  (if (f 3 y) (f x "static") (g x)))
```

By studying the various ways in which `f`, `g`, `x`, and `y` are used in the body of the above abstraction, we can piece together a lot of information about the types of these variables. The application `(f 3 y)`, for example, returns a boolean because it is used as the predicate in an `if` expression. Thus, `f` is a function of two arguments that returns a boolean. From the two calls `(f 3 y)` and `(f x "static")`, we can determine that the the types of `x` and the first argument to `f` are integers, and the type of `y` and the second argument to `f` are strings. The fact that `(f x "static")` and `(g x)` are branches of the same `if` implies that their returns types must be the same. From previous information, we deduce that `g` is a function mapping a single integer parameter to a boolean.

There is no reason that a program cannot carry out the same kind of reasoning exhibited above. Automatically computing the type of an expression that does not contain type information is known as **type reconstruction** or **type inference**. Type reconstruction is more complicated than type checking because type reconstruction must operate properly without programmer supplied type assertions.

Type reconstruction is the formalization of the kind of reasoning seen in the example above. A type reconstruction algorithm is an automatic way of determining the types of an expression (and all the subexpressions along the way). We can think of the different subexpressions in the above example as specifying constraints on the types of the expressions. It is possible to view these constraints as a set of simultaneous type equations that restrict the type of an expression. If these equations can be solved, then the most general typing for the expression results. If these equations cannot be solved, then the expression is not well-typed.

Consider once more `abs` expression studied above. Suppose that:

- τ_{abs} is the type of the result of evaluating the `abs` expression;
- τ_{if} is the type of the result of evaluating the `if` expression;
- τ_{f} is the type of `f`;
- τ_{g} is the type of `g`;
- τ_{x} is the type of `x`;
- τ_{y} is the type of `y`;

Here, subscripted versions of τ are **type variables** that stand for types which we may not yet know. Below are some equations involving these type variables that are implied by our sample HOFL expression:

$\tau_{\text{abs}} = (\rightarrow\ (\tau_{\text{f}}\ \tau_{\text{g}}\ \tau_{\text{x}}\ \tau_{\text{y}})\ \tau_{\text{if}})$ — `abs` *has a function type from args to body*

$\tau_{\text{f}} = (\rightarrow\ (\texttt{int}\ \tau_{\text{y}})\ \tau_{\text{res1}})$ — *In first call to* `f`*, rator has type from arguments to result*

$\tau_{\text{f}} = (\rightarrow\ (\tau_{\text{x}}\ \texttt{string})\ \tau_{\text{res2}})$ — *In second call to* `f`*, rator has type from arguments to result*

$\tau_{\text{g}} = (\rightarrow\ (\tau_{\text{x}})\ \tau_{\text{res3}})$ — *In call to* `g`*, rator has type from arguments to result*

$\tau_{\text{res1}} = \texttt{bool}$ — *Type of* `if` *test must be* `bool`

$\tau_{\text{res2}} = \tau_{\text{res3}}$ — `if` *branches must have same type*

$\tau_{\text{if}} = \tau_{\text{res2}}$ — `if` *has type of first branch*

Above, the types τ_{res1}, τ_{res2}, and τ_{res3} have been introduced as the names of components of τ_{f} and τ_{g}.

A solution to the above equations yields the following variable bindings:

$\tau_{\text{x}} \mapsto \texttt{int}$

$\tau_{\text{y}} \mapsto \texttt{string}$

$\tau_{\text{if}} \mapsto \tau_{\text{res1}} = \tau_{\text{res2}} = \tau_{\text{res3}} = \texttt{bool}$

$\tau_{\text{f}} \mapsto (\rightarrow\ (\texttt{int string})\ \texttt{bool})$

$\tau_{\text{g}} \mapsto (\rightarrow\ (\texttt{int})\ \texttt{bool})$

$\tau_{\text{abs}} \mapsto (\rightarrow\ ((\rightarrow\ (\texttt{int string})\ \texttt{bool})\ (\rightarrow\ (\texttt{int})\ \texttt{bool})\ \texttt{int string})\ \texttt{bool})$

Note that a system of type equations need not always have the neat form of solution indicated by the example. For example, the system associated with

```
(abs (f g x y)
  (if (f 3 4) (f x "static") (g x)))
```

has no solution since it is overconstrained. The first call to `f` implies that `f`'s second argument is an integer, while the second call implies that `f`'s second argument is a string. But integers and strings are disjoint types, so this conflict cannot be resolved.

On the other hand, the system may be underconstrained, as in the following perturbation of the above example:

```
(abs (f g x y)
  (if (f x y) (f x "static") (g x)))
```

In this case, the type of `x` is unknown, and the type deduced for the expression is

$(\rightarrow\ ((\rightarrow\ (\tau_{\text{x}}\ \texttt{string})\ \texttt{bool})\ (\rightarrow\ (\tau_{\text{x}})\ \texttt{bool})\ \tau_{\text{x}}\ \texttt{bool})\ \texttt{bool})$

The appearance of τ_{x} in this type implies that τ_{x} can be instantiated with any type. That is, the type of the abstraction is polymorphic in τ_{x}. In the notation of HOFLEPT, this polymorphic type would be expressed as

```
(forall (t) (→ ((→ (t string) bool) (→ (t) bool) t string) bool))
```

2 Solving Type Equations

In this section, we present the mathematical machinery for solving equations involving types. At the same time, we will show how that machinery can be implemented in SML.

2.1 Type Variables

In order to express and solve type equations, we need a notion of **type variable** that allows us to refer to an unknown type. As shown above, in mathematical notation we will use subscripted versions of τ to stand for type variables.

For creating and manipulating type variables in SML, we will use a `TypeVar` structure matching the `TYPE_VAR` signature in figure 1. A type variable is an entity of abstract type `TVar`. A `TVar` can only be created by calling the nullary `fresh` procedure; every time it is called, it returns a new type variable that is different (as tested by `equal` and `compare`) from any previously created variable. Note that `fresh` does not correspond to a mathematical function because it returns different results when called on the same input. Intuitively, it maintains some sort of local state that allows it to "know" how many times it has been called. It is possible to implement type reconstruction without such "stateful functions", but doing so requires some tedious management of variable names (of the sort seen in the substitution model) that we wish to avoid here.

```
signature TYPE_VAR = sig

  type TVar (* abstract type of type variables *)

  val fresh : unit -> TVar
  (* returns a new type variable not equal to any previously returned *)

  val equal : TVar * TVar -> bool (* test the equality of two type variables *)

  val compare : TVar * TVar -> order (* compare two type variables *)

  val toString : TVar -> string
  (* returns the string representation of a type variable *)

  structure Set : SET
  sharing type Set.item = TVar
  (* an set whose elements type variables *)

  structure Env : ENV
  sharing type Env.key = TVar
  (* an environment whose keys are type variables *)

end
```

Figure 1: SML signature for type variables.

In the particular implementation of `TypeVar` that we shall use, the string representation of a type variable returned by the `toString` function will have the form `#`n, where n is and integer indicating the number of times that `fresh` has been called. The first call to `fresh` will return a type variable with printed representation is `#1`, the second call to `fresh` will return a type variable with printed representation `#2`, and so on.

The `TypeVar` structure contains two component structures: a `Set` structure for modeling sets

whose elements are type variables, and a `Env` structure for modeling environments whose keys are type variables.

In order to manipulate types containing type variables, it is necessary to extend the types of HOFLEMT with an additional constructor `TyVar` for type variables. This is shown in figure 2, which shows the signature for the `Type` structure that will be used in the type reconstructor. The signature also includes a function `TVs` that returns a set of the type variables in a given type and a function `TVsList` that returns a set of the type variables in a given list of types.

```
signature TYPE = sig

  datatype Ty =
    UnitTy | IntTy | BoolTy | SymTy | StringTy (* base types *)
  | ListTy of Ty                               (* component type *)
  | ArrowTy of Ty list * Ty                    (* arg types, result type *)
  (* New for type reconstruction *)
  | TyVar of TypeVar.TVar                      (* type variables *)

  val equal : Ty * Ty -> bool     (* are two types equal? *)
  val toSexpr : Ty -> Sexpr.Sexpr (* S-expression representation of type *)
  val toString : Ty -> string     (* string representation of type *)

  val TVs : Ty -> TypeVar.Set.set (* set of all type variables appearing in type *)
  val TVsList : Ty list  -> TypeVar.Set.set
    (* Set of all type variables appearing in a list of types. *)

end
```

Figure 2: SML signature for types used in type reconstruction.

As an example of manipulating types and type variables, consider implementations of the `TVs` and `TVsList` functions within the `TypeVar` structure, shown in figure 3.

2.2 Type Substitutions

We represent the solution to a set of type equations using an entity called a **type substitution**. A type substitution $\sigma = \bigcup_{i=1}^{n} \{\tau_i \mapsto T_i\}$ is a set of bindings between type variables and types. Here is an example of a type substitution:

$$\sigma_1 = \{\tau_a \mapsto \texttt{int}, \tau_b \mapsto \tau_d, \tau_c \mapsto \texttt{(}\rightarrow\ \texttt{(}\tau_e\ \texttt{bool)}\ \tau_d\texttt{)}\}$$

Note that type variables may appear on the right hand side of a type binding, either by themselves, or as components of other types.

We shall require that all type substitutions satisfy two properties:

1. The type variables in all the bindings of a type substitution are distinct. Formally: if $\tau \mapsto T$ and $\tau' \mapsto T'$ are two bindings in σ, then $T \neq T'$ implies $\tau \neq \tau'$.

```
structure TVSet = TypeVar.Set (* abbreviation *)

fun TVs UnitTy = TVSet.empty
  | TVs IntTy = TVSet.empty
  | TVs BoolTy = TVSet.empty
  | TVs StringTy = TVSet.empty
  | TVs SymTy = TVSet.empty
  | TVs (TyVar(tv)) = TVSet.singleton(tv)
  | TVs (ListTy(ty)) = TVs(ty)
  | TVs (ArrowTy(argTys,resTy)) =
      TVSet.union(TVsList(argTys),
                  TVs(resTy))
 and TVsList tys =
       ListOps.foldr TVSet.union TVSet.empty (map TVs tys)
```

Figure 3: Functions for calculating the type variables appearing in a type or list of types.

This condition implies that a type substitution can be viewed as a partial function from type variables to types. A partial function is one that may be defined for some type variables and undefined for others. The type variables on which the partial function is defined is called the **domain** of the partial function; in this case, the domain of a type substitution $\sigma = \bigcup_{i=1}^{n}\{\tau_\mathrm{i} \mapsto T_\mathrm{i}\}$ is $dom(\sigma) = \bigcup_{i=1}^{n}\{\tau_\mathrm{i}\}$.

Because type substitutions can be viewed as functions, we will use mathematical function application notation $\sigma(\tau)$ to denote the type bound to a type variable in σ. The **codomain** of a substitution is the set of all types bound to type variables in its domain: $cod(\sigma) = \bigcup_{i=1}^{n}\{\sigma(\tau_\mathrm{i})\}$.

2. No type bound to a type variable in a type substitution mentions one of the bound type variables of the type subsitution. Formally: $dom(\sigma) \bigcap (\bigcup_{\tau\in\sigma} TVs(\sigma(\tau))) = \{\}$

   A type T is **canonical** with respect to (w.r.t.) a set S of type variables if it does not contain them (i.e., $S \bigcap TVs(T) = \{\}$). By extension, a type is canonical w.r.t. a type substitution σ if it is canonical w.r.t. $dom(\sigma)$. So another way to phrase the second property of substitutions is that each type bound in a type substitution σ must be canonical w.r.t. σ. We can also "lift" the canonical notion to expressions: an expression E is canonical w.r.t. a set of type variables S (resp. type substitution σ) if it does not contain any type variables in S (resp. $dom(\sigma_)$). The notion can similarly be lifted to type environments: an type environment A is canonical w.r.t. a set of type variables S (resp. type substitution σ) if $cod(A)$ does not contain any type variables in S (resp. $dom(\sigma_)$).

Given that type substitutions can be viewed as functions on type variables, it is natural to "lift" them to act as functions on types. Applying a type substitution σ to a type T yields a copy of T in which all occurrences of any $\tau \in dom(\sigma)$ has been replaced by $\sigma(\tau)$. We will abuse notation and write $\sigma(T)$ for the result of applying σ to T. For example, suppose that T_1 is:

$$(\rightarrow\ (\tau_\mathrm{a}\ \texttt{(listof}\ \tau_\mathrm{c}\texttt{))}\ (\rightarrow\ (\tau_\mathrm{d}\ \tau_\mathrm{f})\ \tau_\mathrm{b})).$$

Then $\sigma_1(T_1)$ is:

$$(\rightarrow \texttt{ (int (listof (}\rightarrow \texttt{ (}\tau_{\mathrm{e}}\texttt{ bool) }\tau_{\mathrm{d}}\texttt{))) (}\rightarrow \texttt{ (}\tau_{\mathrm{d}}\texttt{ }\tau_{\mathrm{f}}\texttt{) }\tau_{\mathrm{d}}\texttt{)).}$$

Note that $\sigma(\tau) = \tau$ if $\tau \notin dom(\sigma)$.

We can similarly lift application of a type substitution to expressions, and will write $\sigma(E)$ for a copy of E in which every occurrence of a $\tau \in dom(\sigma)$ has been replaced by $\sigma(\tau)$. For example, suppose that the expression E is:

```
(abs ((f τc) (x τa) (y τb))
  (f (prepend x (empty τa)) (= x y)))
```

Then the result of applying σ to E, written $\sigma(E)$, is:

```
(abs ((f (→ (τe bool) τd)) (x int) (y τd))
  (f (prepend x (empty int)) (= x y)))
```

It is also possible to lift the application of type substitutions to type environments, where a type environment A can be viewed as function from HOFLIMT identifiers to HOFLEMT types. In this case, $(\sigma(A))(I) = \sigma(A(I))$.

Note that applying a type substitution σ to any type T, expression E, or type environment A replaces all occurrences of type variables in $dom(\sigma)$ by types that (by property 2 of type substitutions) do not contain occurrences of these type variables. Therefore, $\sigma(T)$, $\sigma(E)$, and $\sigma(A)$

In the SML implementation, type substitutions are elements of the type `subst`. Applying a type substitution to types and expressions is accomplished functions with the following signatures:

```
val subTy : subst -> Type.Ty -> Type.Ty
val subTys : subst -> Type.Ty list -> Type.Ty list
val subExp : subst -> AST.Exp -> AST.Exp
val subExps : subst -> AST.Exp list -> AST.Exp list
```

The implementation of these functions is shown in figure 4. Type substitutions are implemented as environments that are keyed by type variables (i.e., elements of `TypeVar.Env`). Applying these type substitutions to types and expressions is performed by straightforward recursive tree walks.

2.3 Unification

Suppose that we are given a type equation that equates two types, each of which may contain type variables. For example, we might be given $T_{\mathrm{a}} = T_{\mathrm{b}}$, where

$T_{\mathrm{a}} \equiv$ (→ (τ_{a} (listof τ_{b})) int)
$T_{\mathrm{b}} \equiv$ (→ (bool τ_{c}) τ_{b}).

```
structure SubstEnv = TypeVar.Env
type subst = Type.Ty SubstEnv.env

and subTy s UnitTy = UnitTy
  | subTy s IntTy = IntTy
  | subTy s BoolTy = BoolTy
  | subTy s StringTy = StringTy
  | subTy s SymTy = SymTy
  | subTy s (typ as TyVar(tv)) =
     (case SubstEnv.lookup(tv,s) of
       NONE => typ
      | SOME(t) => t
        (* By subst property 2, t is canonical w.r.t. s
           and so no further substitutions must be performed *)
      )
  | subTy s (ListTy(eltTy)) = ListTy(subTy s eltTy)
  | subTy s (ArrowTy(argTys, resTy)) =
      ArrowTy(subTys s argTys,subTy s resTy)

and subTys s tys = List.map (subTy s) tys

and subExp s (lit as Lit(_)) = lit
  | subExp s (vref as VarRef(_)) = vref
  | subExp s (PrimApp(primop,rands)) =
     PrimApp(primop,subExps s rands)
  | subExp s (PrimEmpty(ty)) = PrimEmpty(subTy s ty)
  | subExp s (If(test,thenExp,elseExp)) =
     If(subExp s test, subExp s thenExp, subExp s elseExp)
  | subExp s (Abs(formals,types,body)) =
     Abs(formals, subTys s types, subExp s body)
  | subExp s (FunApp(rator,rands)) =
     FunApp(subExp s rator, subExps s rands)
  | subExp s (BindPar(names,defns,body)) =
     BindPar(names,subExps s defns, subExp s body)
  | subExp s (BindRec(names,types,defns,body)) =
     BindRec(names, subTys s types, subExps s defns, subExp s body)

and subExps s exps = List.map (subExp s) exps
```

Figure 4: Implementation of functions for applying type substitutions to types and expressions.

(The notation $T \equiv T'$ means that T and T' are syntactically identical.) We would like to be able to "solve" such a type equation. In algebra, solving an equation with variables means finding a substitution for the variables that makes both sides of the equation the same value or expression. The same is true for type equations: a solution to a type equation $T_1 = T_2$ is a type substitution σ such that $\sigma(T_1) \equiv \sigma(T_2)$. In the above example, a solution to $T_\mathrm{a} = T_\mathrm{b}$ is the following substitution:

$$\sigma = \{\tau_\mathrm{a} \mapsto \texttt{bool}, \tau_\mathrm{b} \mapsto \texttt{int}, \tau_\mathrm{c} \mapsto \texttt{(listof int)}\}$$

We can verify that $\sigma(T_\mathrm{a}) \equiv \sigma(T_\mathrm{b}) \equiv$ `(→ (bool (listof int)) int)`.

A type substitution that makes the two types in a type equation syntactically equal is called a **unifier** for the two types. If a unifier exists, we say that the two types can be **unified** and that the unifier is the result of the **unification** of the two types. Not every pair of types can be unified; in the case where no unifier exists, we say that unification **fails**. For instance, the type equation `int` = `bool` is clearly insoluble, as is `(→ (`τ`) int)` = `(→ (bool) `τ`)`, so unification fails in both of these cases. A trickier case is $\tau =$ `(listof `τ`)`. Here there is no finite type T such that $\{\tau \mapsto T\}$ is a unifier, so the equation is insoluable. (If we allowed infinite types, then the infinitely nested list type $T =$ `(listof (listof (listof ...)))` would be a solution.) By the same reasoning, any type equation of the form $\tau = T$ where $\tau \in TVs(T)$ is insoluable. In this case, unification is said to fail because of the "occurs check" $\tau \in TVs(T)$.

In some cases there may be many solutions to a type equation. For instance, consider the following type equation:

$$\texttt{(→ (}\tau_1\texttt{) }\tau_2\texttt{)} = \texttt{(→ (}\tau_2\texttt{) }\tau_3\texttt{)}$$

This equation can be solved by any unifier that binds τ_1, τ_2, and τ_3 to the same type. Examples of such unifiers are:

$$\begin{aligned}
\sigma_1 &= \{\tau_1 \mapsto \texttt{int},\ \ \tau_2 \mapsto \texttt{int},\ \ \tau_3 \mapsto \texttt{int}\} \\
\sigma_2 &= \{\tau_1 \mapsto \texttt{bool},\ \ \tau_2 \mapsto \texttt{bool},\ \ \tau_3 \mapsto \texttt{bool}\} \\
\sigma_3 &= \{\tau_1 \mapsto \tau_3,\ \ \tau_2 \mapsto \tau_3\} \\
\sigma_4 &= \{\tau_1 \mapsto \tau_2,\ \ \tau_3 \mapsto \tau_2\} \\
\sigma_5 &= \{\tau_1 \mapsto \tau_4,\ \ \tau_2 \mapsto \tau_4,\ \ \tau_3 \mapsto \tau_4\}
\end{aligned}$$

(The canonicalization property of type substitutions prevents $\tau_3 \mapsto \tau_3$ from being explicitly listed in σ_3 and $\tau_2 \mapsto \tau_2$ from being explicitly listed in σ_4. But, when viewed as functions on types, σ_3 still maps τ_3 to τ_3 and σ_4 still maps τ_2 to τ_2.)

Intuitively, the substitutions σ_3, σ_4, and σ_5 are more general than σ_1 and σ_2, because the latter can be obtained from the former by instantiating more type variables. For instance, σ_1 can be obtained from σ_5 by instantiating τ_4 to `int`. We shall say that σ_1 is more general than σ_2 (written $\sigma_1 \geq \sigma_2$) if there exists an **instantiation substitution** σ_inst such that $\sigma_\mathrm{inst} \circ \sigma_1 = \sigma_2$. Here, $\circ$ is function composition; i.e., $(\sigma_\mathrm{inst} \circ \sigma_1)(T) = \sigma_\mathrm{inst}(\sigma_1(T))$. It turns out that if HOFLEMT types T_1 and T_2 have a unifier, then they have a **most general unifier** σ_mgu such that $\sigma_\mathrm{mgu} \geq \sigma$ for every unifier σ of T_1 and T_2. In the above example, σ_3, σ_4, and σ_5 are all most general unifiers for the two types in the equation.

2.4 A Unification Algorithm

Here we present a unification algorithm that, given two types, returns their most general unifier if it exists, and otherwise indicates that unification fails for the two types. We shall express the algorithm in SML using the following signature:

```
type subst
exception UnifyError of string
val empty: subst
val unify : (Type.Ty * Type.Ty) -> subst
val unifyList : (Type.Ty list * Type.Ty list) -> subst
val union : (subst * subst) -> subst
val unionList : subst list -> subst
val subTy : subst -> Type.Ty -> Type.Ty
val subTys : subst -> Type.Ty list -> Type.Ty list
val subExp : subst -> AST.Exp -> AST.Exp
val subExps : subst -> AST.Exp list -> AST.Exp list
```

The type `subst` is the SML type of type substitutions, and `empty` is the empty type substitution. Given two types, `unify` either returns an element of type `subst` that is the most general unifier of the two types or raises a `UnifyError` exception if the two types cannot be unified. The `unifyList` function returns the most general unifier that unifies corresponding pairs of types in two lists of types. The `union` function returns a single substitution that combines all type constraint information in a single substitution. The `unionList` function combines the type constraint information in a list of substitutions into a single substitution. The `unifyList`, `union`, and `unionList` functions all raise a `UnifyError` exception if the type constraints cannot be solved. The `subTy`, `subTys`, `subExp`, and `subExps` functions were described in section 2.2.

As in section 2.2, we shall assume that substitutions are represented as environments of type `SubstEnv` that map type variables to types (i.e., `SubstEnv` is an abbreviation for `Type.Ty TypeVar.Env`). Such environments have the environment signature in figures 11–12 where `TypeVar.TyVar` replaces the `key` type and `'b` is instantiated to the type `Type.Ty`.

An SML implementation of the above unification signature is presented in figures 5–6. The `unify` implementation is relatively straightforward. Equal base types can be unified with an empty substitution, and compound types with the same type constructor (e.g., `listof` or `->`) can be unified by recursively unifying their corresponding components. The most important case is when a type variable τ is being unified to a type T. In this case, an occurs check is performed to see if τ appears in T; if not, a substitution with the single binding $\{\tau \mapsto T\}$ is returned.

The case of unifying a type variable τ with itself needs to be handled specially. Such a unification should always succeed, but a naively applying the occurs check would cause it to fail. In this case, the implementation returns a substitution containing an identity binding $\{\tau \mapsto \tau\}$. Technically, this violates the canonicalization property of type substitutions, since the right-hand side constains a type variable bound on the left-hand side. However, when a type substitution is viewed as a function on types, type environments, or expressions, a type substitution containing an identity binding is behaviorally indistinguishable from one without the binding. Because it is convenient and not harmful, the implementation allows substitutions to contain such identity bindings, but

```
structure SubstEnv = TypeVar.Env
type subst = Type.Ty SubstEnv.env

exception UnifyError of string

val empty = SubstEnv.empty

fun unify(UnitTy,UnitTy) = empty
  | unify(IntTy,IntTy) = empty
  | unify(BoolTy,BoolTy) = empty
  | unify(StringTy,StringTy) = empty
  | unify(SymTy,SymTy) = empty
  | unify(ListTy(t1),ListTy(t2)) = unify(t1,t2)
  | unify(ArrowTy(argTys1,resTy1),ArrowTy(argTys2,resTy2)) =
      unifyList(resTy1::argTys1,resTy2::argTys2)
  | unify(TyVar(tv1), t2) = unifyTVar(tv1,t2)
  | unify(t1, TyVar(tv2)) = unifyTVar(tv2,t1)
  | unify(t1,t2) =
      raise UnifyError("Cannot unify the following types:\n"
                       ^ (Type.toString(t1)) ^ "\n"
                       ^ (Type.toString(t2)) ^ "\n")

and unifyTVar(tv,typ) =
    let val _ = occursCheck(tv,typ)
     in SubstEnv.bind(tv,typ,empty)
    end

and occursCheck(tv,typ) =
    if occursIn(tv,typ) andalso
       (not (sameTypeVar(tv,typ)))
       (* It's OK for a type variable to be bound to itself *)
    then raise UnifyError ("Occurs check failed!\n" ^ "The type variable "
                           ^ (TypeVar.toString(tv)) ^ " occurs in the type:\n"
                           ^ (Type.toString(typ)) ^ "\n")
    else ()

and occursIn(tv,typ) = TypeVar.Set.member(Type.TVs(typ),tv)

and sameTypeVar(tv, TyVar(tv')) = TypeVar.equal(tv,tv')
  | sameTypeVar(tv, _) = false

and unifyList(ts1,ts2) =
    if length(ts1) = length(ts2) then
      foldr union empty (ListOps.map2 unify ts1 ts2)
    else
      raise UnifyError ("unifyList: mismatch in length of lists:\n"
                       ^ (typesToString(ts1)) ^ "\n"
                       ^ (typesToString(ts2)) ^ "\n")
```

Figure 5: Implementation of unification, part 1.

enforces the canonicalization property for all non-identity bindings.[1]

Combining two type substitutions σ_1 and σ_2 via `union` is the the trickiest part of the unification implementation. While σ_1 and σ_2 both necessarily satisfy the canonicalization property, we must ensure that their combination does as well. Even in the case where $dom(\sigma_1) \bigcap dom(\sigma_2) = \{\}$, it is not correct to simply take the union of the bindings of the two substitutions, since it may be that types in $cod(\sigma_1)$ reference types in $dom(\sigma_2)$ or that types in $cod(\sigma_2)$ reference types in $dom(\sigma_1)$. For example, consider:

$$\sigma_a = \{\tau_1 \mapsto \tau_2,\ \ \tau_3 \mapsto \tau_4,\ \ \tau_5 \mapsto \tau_6\}$$
$$\sigma_b = \{\tau_2 \mapsto \tau_3,\ \ \tau_4 \mapsto \tau_5,\ \ \tau_6 \mapsto \tau_{int}\}$$

Unioning σ_a and σ_b should yield a substitution in which all six type variables are bound to `int` – a result that clearly cannot be obtained with a straightforward union!

Moreover, combining substitutions with disjoint domains can introduce cycles where none existed before. For example, consider:

$$\sigma_c = \{\tau_3 \mapsto \tau_4\}$$
$$\sigma_d = \{\tau_4 \mapsto \texttt{(listof } \tau_3\texttt{)}\}$$

Unioning σ_c and σ_d should fail because a binding of τ_3 to `(listof` τ_3`)` violates the occurs check.

In the case where $S = dom(\sigma_1) \bigcap dom(\sigma_2)$ is non-empty, it is additionally necessary to incorporate into the result any constraints from unifying $\sigma_1(\tau)$ and $\sigma_2(\tau)$ for all $\tau \in S$.

The implementation of `union` in figure 6 handles all of these situations. It works by collecting a result substitution by using an `insert` function to insert the bindings of σ_1 one at a time into a growing result substitution that is initialized to σ_2. When inserting the binding $\tau \mapsto T$ into the answer substitution σ_{ans}, `insert` needs to handle two cases:

1. If $\tau \notin dom(\sigma_{ans})$,

   - The type T is canonicalized with respect to σ_{ans} to create $T' = \sigma_{ans}(T)$. Since T' does not contain any type variables in $dom(\sigma_{ans})$, the binding $\tau \mapsto T'$ can extend the answer substitution *as long as* T' also does not contain τ itself. The occurs check performed by `unifyTVar` guarantees this.
   - Since types in $cod(\sigma_{ans})$ may contain τ, it is necessary to map the application of the substitution $\tau \mapsto T'$ over each type in $cod(\sigma_{ans})$. This removes all occurrences of τ from the right-hand sides of substitutions (making all of them canonical with respect to the new binding $\tau \mapsto T'$) and preserves the canonical property of each type in $cod(\sigma_{ans})$ w.r.t. $dom(\sigma_{ans})$ (since T' is canonical w.r.t. $dom(\sigma_{ans})$).

2. If $\tau \in dom(\sigma_{ans})$,

   - The type T is canonicalized with respect to σ_{ans} to create $T' = \sigma_{ans}(T)$. In this case, since $\tau \in dom(\sigma_{ans})$, we know that T' cannot contain τ, and so no occurs check is necessary.

[1] Although identity bindings would be easy to remove in `unifyTVar`, they would be harder to remove in the `insert` function in figure 6.

```
and union(s1,s2) = SubstEnv.foldri insert s2 s1

and insert(tv,typ,ans) =
  let val typ' = subTy ans typ
   in case SubstEnv.lookup(tv,ans) of
        NONE =>
          SubstEnv.bind(tv,typ',SubstEnv.map (subTy (unifyTVar(tv,typ'))) ans)
      | SOME(ty) =>
         let val newsub = unify(typ',ty)
          in SubstEnv.foldri SubstEnv.bind
                             (SubstEnv.map (subTy newsub) ans)
                             newsub
         end
  end

and unionList(substs) = foldr union empty substs
```

Figure 6: Implementation of unification, part 2.

- A substitution σ_{new} is created by unifying T' and $\sigma_{\text{ans}}(\tau)$. It is guaranteed that $dom(\sigma_{\text{ans}})$ is disjoint from $dom(\sigma_{\text{new}})$ and any type variables in $cod(\sigma_{\text{new}})$.
- The bindings of σ_{new} can be unioned with those of σ_{ans} as long as the types in $cod(\sigma_{\text{ans}})$ are first canonicalized w.r.t. σ_{new}. This is accomplished by mapping the application of σ_{new} over each type in $cod(\sigma_{\text{ans}})$.

3 Monomorphic Type Reconstruction

In this section we show how to use the unification technology developed in the previous section to implement type reconstruction for a language with a monomorphic type system. The type reconstruction algorithm shown here, due to Hindley and Milner, is at the core of type reconstruction in modern languages like ML and Haskell. The Hindley-Milner algorithm can also reconstruct types for a restricted class of polymorphic functions, though we shall not study this important feature here.

3.1 HOFLIMT

We illustrate type reconstruction in the context of automatically deriving an explicitly typed HOFLEMT program from a program in an implicitly typed language we shall christen HOFLIMT. The HOFLIMT language has exactly the same syntax as the dynamically typed HOFL language. The difference between HOFLIMT and HOFL is that a HOFLIMT program is required to be **typable**, in the sense that it can be transformed into a well-typed HOFLEMT program simply by adding type annotations without making any other changes.

For example, consider the three HOFL programs in figure 7. The first of these programs (P_1) is a HOFLIMT program because it can be annotated to be the following well-typed HOFLEMT program:

```
(program (a b)
  (bindpar ((appa (abs ((f (-> (int) int))) (f a)))
            (inc (abs ((x int)) (+ x 1)))
            (pos (abs ((y int)) (> y 0))))
    (prepend a
      (prepend (if (pos a) (appa inc) b)
        (empty int)))))
```

However, neither P_2 nor P_3 is a HOFLIMT program. In P_2, `appa` is used polymorphically – it is applied to both `pos` (a function of type `(→ (int) int)`) and `inc` (a function of type `(→ (int) bool)`). There is no way to give the single copy of `app5` a type that matches both of these uses. In P_3, `appa` is used monomorphically, but an attempt is made to make a list with both boolean and integer elements – something that is fine in HOFL, but cannot be type-checked in HOFLEMT.

$P_1 \equiv$

```
(program (a b)
  (bindpar ((appa (abs (f) (f a)))
            (inc (abs (x) (+ x 1)))
            (pos (abs (y) (> y 0))))
    (prepend a
      (prepend (if (pos a) (appa inc) b)
        (empty)))))
```

$P_2 \equiv$

```
(program (a b)
  (bindpar ((appa (abs (f) (f a)))
            (inc (abs (x) (+ x 1)))
            (pos (abs (y) (> y 0))))
    (prepend a
      (prepend (if (appa pos) (appa inc) b)
        (empty)))))
```

$P_3 \equiv$

```
(program (a b)
  (bindpar ((appa (abs (f) (f a)))
            (inc (abs (x) (+ x 1)))
            (pos (abs (y) (> y 0))))
    (prepend (pos a)
      (prepend (appa inc)
        (empty)))))
```

Figure 7: Sample HOFL programs. Only P_1 is a HOFLIMT program.

Note that any well-typed HOFLEMT program can be transformed into an implicitly typed HOFLIMT program simply by erasing the type annotations. Although we shall not do so here, it is possible to formalize the well-typedness of HOFLIMT programs by using versions of the HOFLEMT typing rules in which all the type annotations on expressions have been erased.

3.2 Type Reconstruction For HOFLIMT

The SML implementation of a type reconstruction algorithm for HOFLIMT appears in figures 8–10. The entry point to the type reconstructor is the following function:

```
val reconProg: Hofl.AST.Program -> AST.Program
```

The `Hofl.AST` structure implements the abstract syntax tress of HOFLIMT (which are exactly the same as those of HOFL). The `reconProg` function transforms a program from this structure to one in the `AST` structure, which implements the abstract syntax trees for a version of HOFLEMT in which types have been extended (as in Section 2.1) to include type variables. The type reconstructor must use explicitly qualified names for abstract syntax nodes like `Abs`, `FunApp`, `If`, etc., to distinguish whether they are HOFLIMT nodes or HOFLEMT nodes. The following structure abbreviations are introduced to make this distinction more concise

```
structure I = Hofl.AST
structure E = AST
```

Here `I` stands for the syntax in the **i**mplicitly typed language, while `E` stands for the syntax in the **e**xplicitly typed language. The **i** and **e** letters are also used in variable names within the code as prefixes that indicate which expressions denote implicitly typed syntax and which denote explicilty typed syntax.

The core of the type reconstruction algorithm are the `reconExp` and `reconExps` functions, whose signatures are:

```
val reconExp: Hofl.AST.Exp
              -> Type.Ty Ident.Env.env
              -> (AST.Exp * Type.Ty * Subst.subst)

val reconExps: Hofl.AST.Exp list
              -> Type.Ty TypeVar.Env.env
              -> (AST.Exp list * Type.Ty list * Subst.subst)
```

The `reconExp` function takes as arguments (1) a HOFLIMT expression E_{I} and (2) a type environment A_{I} mapping free expression variables to their types. Its result is a triple with the following components:

- A well-typed HOFLEMT expression E_{E} whose type erasure is E_{I}.
- The type T_{E} of E_{E}.
- A type substitution σ_{E} that contains solutions to type variable constraints encountered during the type reconstruction of E_{I}.

The elements of the triple satisfy the following type judgement:

$$\sigma_{\mathrm{E}}(A_{\mathrm{I}}) \vdash \sigma_{\mathrm{E}}(E_{\mathrm{E}}) : \sigma_{\mathrm{E}}(T_{\mathrm{E}})$$

```
structure I = Hofl.AST (* Implicitly typed syntax *)
structure E = AST      (* Explicitly typed syntax *)
structure TEnv = Ident.Env

exception ReconError of string

fun newTyVar () = TyVar(TypeVar.fresh())

fun reconProg (I.Prog(formals,ibody)) =
    let val (ebody, _, sub) =
          reconExp ibody (TEnv.extend(formals,
                                      map (fn _ => IntTy) formals,
                                      TEnv.empty))
     in E.Prog(formals,subExp sub ebody)
    end

and reconExp (I.Lit(lit)) tenv = (E.Lit(lit), litType(lit), Subst.empty)

  | reconExp (I.VarRef(var)) tenv =
      (case TEnv.lookup(var, tenv) of
         NONE => raise ReconError("Unbound variable: " ^
                                  (Ident.toString(var)))
       | SOME(ty) => (E.VarRef(var), ty, Subst.empty)
       )

  | reconExp (I.Abs(formals,ibody)) tenv =
      let val formalTys = map (fn _ => newTyVar()) formals
          val (ebody, bodyTy, bodySub) =
                reconExp ibody (TEnv.extend(formals,formalTys,tenv))
       in (E.Abs(formals, formalTys, ebody),
           ArrowTy(formalTys,bodyTy),
           bodySub)
      end

  | reconExp (I.FunApp(irator, irands)) tenv =
      let val (erator, ratorTy, ratorSub) = reconExp irator tenv
          val (erands, randTys, randsSub) = reconExps irands tenv
          val resultTy = newTyVar()
          val appSub = unionList[unify(ratorTy, ArrowTy(randTys, resultTy)),
                                 ratorSub,
                                 randsSub]
       in (E.FunApp(erator,erands), resultTy, appSub)
      end
```

Figure 8: Core type reconstruction, part 1.

The `reconExps` function is similar to `reconExp` except that it takes a list of expressions rather than a single expression, and the triple it returns contains a list of expressions and a list of types. The third component of the resulting triple is a single type substitution (not a list of such substitutions) that combines constraint information collected from reconstructing all of the expressions.

Figures 8–9 show the implementation of `reconProg`, `reconExp`, and `reconExps`. `reconProg` invokes `reconExp` on the program body in a type environment that assumes all program formals are integers. The literal and variable cases for `reconExp` are straightforward. Neither introduces any type variable constraints, so the returned substitution is empty in both cases.

The abstraction and application cases are more interesting; in some sense, they are the heart of the algorithm. When we encounter an abstraction, we have no idea what the types of the formal parameters should be, so we introduce a fresh type variable for each parameter into the type environment for reconstructing the abstraction body. This type variable will be involved in any type constraints implied by uses of the parameter in the body. The explicitly typed abstraction that is returned annotates each formal parameter with its associated type variable, and the returned type is the function type expected from HOFLEMT's typing rule for abstractions. The returned substitution is the one that results from reconstructing the abstraction's body. This substitution will be combined with constraints accumulated in the rest of the program, and the final program substitution will be used to resolve (when possible) to type variables introduced for each formal parameter.

The `FunApp` case is the first case of `reconExp` that introduces type variable constraints. Given the implicitly typed function application $(E_{\mathrm{I}0} \ \ldots \ E_{\mathrm{In}})$, the clause recursively reconstructs the subexpressions to yield the explicitly type expressions $E_{\mathrm{E}0} \ \ldots \ E_{\mathrm{En}}$, their types $T_{\mathrm{E}0} \ \ldots \ T_{\mathrm{En}}$, and the substitution σ_{E}. The typing rule for applications is asserted by unifying the operator type $T_{\mathrm{E}0}$ with the function type $(\rightarrow \ (T_{\mathrm{E}1} \ \ldots T_{\mathrm{En}}) \ \tau_{\mathrm{res}})$, where τ_{res} is a freshly generated type variable denoting the result type of the application. Any constraints determined by the unification process are merged with σ_{E} to yield the substitution `appSub` for the abstraction. These constraints will later be used to resolve any occurences of τ_{res} in the reconstructed program.

The `if` case also introduces constraints. It uses unification to guarantee that the test expression has type `bool` and that the then and else branch have the same type, which is returned a the type of the `if`.

Primitive applications are handled similarly to function applications. The `primArrowTy` function (detailed in figure 10) returns an arrow type for each primop, at which point this case effectively becomes equivalent to a function application. An important feature of `primArrowTy` is that it returns freshly generated type variables for each invocation of a list operation. This allows list operations to be treated polymorphically. The fact that the empty list is encoded via special syntax in HOFLEMT (`PrimEmpty`) means that the empty list primop must be handled specially.

The final case of `reconExp` is `BindRec`. To handle the recursive scope of the bound identifiers, the reconstructor creates a type Genvironment `recTenv` that extends the given type environment with a binding between each bound name and a fresh type variable. It then ties the recursive knot by unifying the freshly chosen type variables with the resulting definition types, and finally reconstructs the `bindrec` body relative to the extended environment.

A Environment Signature

```
      | reconExp (I.PrimApp(primop,irands)) tenv =
          let val (erands,randTys,randsSub) = reconExps irands tenv
              val resTy = newTyVar()
              val primSub = union(unify(primArrowTy(primop),
                                        ArrowTy(randTys,resTy)),
                                  randsSub)
              val resTy' = subTy primSub resTy
           in (case primop of
                 (* Handle Empty specially *)
                 Empty => (case resTy' of
                             ListTy(t) => E.PrimEmpty(t)
                           | _ => raise ReconError
                                        ("Shouldn't happen: non list type for empty")
                           )
               | _ => E.PrimApp(primop,erands),
               resTy',
               primSub)
          end

      | reconExp (I.If(itest,ithen,ielse)) tenv =
          let val (etest,testTy,testSub) = reconExp itest tenv
              val (ethen,thenTy,thenSub) = reconExp ithen tenv
              val (eelse,elseTy,elseSub) = reconExp ielse tenv
              val ifSub = unionList[unify(testTy, BoolTy),
                                    unify(thenTy,elseTy),
                                    testSub,
                                    thenSub,
                                    elseSub]
           in (E.If(etest,ethen,eelse), thenTy, ifSub)
          end

      | reconExp (I.BindRec(names,idefns,ibody)) tenv =
          let val defnTyVars = map (fn _ => newTyVar()) idefns
              val recTenv = TEnv.extend(names, defnTyVars, tenv)
              val (edefns, defnTys, defnsSub) = reconExps idefns recTenv
              val (ebody, bodyTy, bodySub) =
                    reconExp ibody (TEnv.extend(names, defnTys, recTenv))
              val recSub = unionList[unifyList(defnTyVars, defnTys),
                                     defnsSub,
                                     bodySub]
           in (E.BindRec(names, defnTyVars,edefns,ebody), bodyTy, recSub)
          end

    and reconExps iexps tenv =
        let val (eexps,tys,subs) =
              ListOps.unzip3(map (fn exp => reconExp exp tenv) iexps)
         in (eexps,tys,unionList(subs))
        end
```

Figure 9: Core type reconstruction, part 2.

```
val arithopTy = ArrowTy([IntTy,IntTy],IntTy)
val relopTy = ArrowTy([IntTy,IntTy],BoolTy)
val logopTy = ArrowTy([BoolTy,BoolTy],BoolTy)

fun primArrowTy(Add) = arithopTy
  | primArrowTy(Sub) = arithopTy
  | primArrowTy(Mul) = arithopTy
  | primArrowTy(Div) = arithopTy
  | primArrowTy(Mod) = arithopTy
  | primArrowTy(LT) = relopTy
  | primArrowTy(LE) = relopTy
  | primArrowTy(EQ) = relopTy
  | primArrowTy(NE) = relopTy
  | primArrowTy(GE) = relopTy
  | primArrowTy(GT) = relopTy
  | primArrowTy(Band) = logopTy
  | primArrowTy(Bor) = logopTy
  | primArrowTy(Not) = ArrowTy([BoolTy],BoolTy)
  | primArrowTy(Symeq) = ArrowTy([SymTy,SymTy],BoolTy)

  (* For the following, return new type vars for every call *)
  | primArrowTy(Empty) = ArrowTy([],ListTy(newTyVar()))
  | primArrowTy(IsEmpty) = ArrowTy([ListTy(newTyVar())], BoolTy)
  | primArrowTy(Head) =
      let val t = newTyVar()
       in ArrowTy([ListTy(t)], t)
      end
  | primArrowTy(Tail) =
      let val t = newTyVar()
       in ArrowTy([ListTy(t)], ListTy(t))
      end
  | primArrowTy(Prepend) =
      let val t = newTyVar()
       in ArrowTy([t, ListTy(t)], ListTy(t))
      end

fun litType(UnitLit) = UnitTy
  | litType(IntLit(_)) = IntTy
  | litType(BoolLit(_)) = BoolTy
  | litType(StringLit(_)) = StringTy
  | litType(SymLit(_)) = SymTy
```

Figure 10: Auxiliary functions used by type reconstruction algorithm.

```
signature STRING_ENV = sig

    type key = string

    type 'b env
    (* Type of env that maps string keys to values of type 'b *)

    val empty : 'b env
    (* empty denotes an empty env *)

    val bind : (string * 'b * 'b env) -> 'b env
    (* bind(key,value,tbl) returns a new env that includes the
       binding key->value in addition to  all bindings of tbl.
       Any existing binding for key in tbl is shadowed by key->value. *)

    val extend : (string list * 'b list * 'b env) -> 'b env
    (* extend(keys,values,tbl) returns a new env that includes
       corresponding bindings between keys and values in addition to
       all bindings of tbl. Any  existing binding for keys in tbl
       are shadowed by the new bindings. *)

    val lookup : (string * 'b env) -> 'b option
    (* lookup(key,tbl) returns SOME(value) if tbl contains the binding
       key->value. If there is no binding for key, lookup returns NONE. *)

    val unbind : (string * 'b env) -> 'b env
    (* unbind(key,tbl) returns a new env that includes all
       bindings of tbl except for a binding for key. *)

    val remove : (string list * 'b env) -> 'b env
    (* unbind(keys,tbl) returns a new env that includes all
       bindings of tbl except for bindings for keys. *)

    val bindingsToEnv : (string * 'b) list -> 'b env
    (* bindingsToEnv(keyValuePairs) returns a env whose bindings consist
       of all the bindings specified by the list of pairs keyValuePairs. *)

    val keys : 'b env -> string list
    (* keys(tbl) returns a list of all keys for bindings in tbl.
       Each key is mentioned only once. *)

    val values : 'b env -> 'b list
    (* values(tbl) returns a list of all values for bindings in tbl.
       The values are in the same order as the keys returned by
       keys(tbl). Because the same value may be bound to more than
       one key, the result may contain duplicates. *)
```

Figure 11: Environment signature, part 1.

```
  val map : ('a -> 'b) -> 'a env -> 'b env
  (* (map f env) returns an environment of bindings {k -> f(v) |
     (k -> v)} is a binding in env. *)

  val mapi : ((key * 'a) -> 'b) -> 'a env -> 'b env
  (* (map f env) returns an environment of bindings {k -> f(k,v) |
     (k -> v)} is a binding in env. *)

  val foldr : (('a * 'b) -> 'b) -> 'b -> 'a env -> 'b
  (* (foldr f z env) returns the result of folding f from right-to-left
     starting with z over the values of env (ordered by key). *)

  val foldri : ((key * 'a * 'b) -> 'b) -> 'b -> 'a env -> 'b
  (* Like foldr, but f takes the key as well as the value and answer *)

  val foldl : (('a * 'b) -> 'b) -> 'b -> 'a env -> 'b
  (* Like foldr, but accumulates values left-to-right *)

  val foldli : ((key* 'a * 'b) -> 'b) -> 'b -> 'a env -> 'b
  (* Like foldl, but f takes the key as well as the value and answer *)
end
```

Figure 12: Environment signature, part 2.