Types

1 Static Properties of Programs

This is a change we’re making as a test.

Programs have both dynamic and static properties:

- A **dynamic property** is one that can be determined in general only at run-time by executing the program.

- A **static property** is one that can be determined without executing the program. Static properties are often determined at compile time by a compiler.

For instance, consider the following Scheme expression:

```scheme
(let ((n (read)) ; Scheme’s READ reads a value from user
     (sq (lambda (x) (* x x)))
     (if (integer? n)
         (+ (sq (- n 1)) (sq (+ n 1)))
         0))
```

The value of this expression is a dynamic property, because it cannot be known until run-time what input will be entered by the user. However, there are numerous static properties of this program that can be determined at compile-time:

- The free variables of the expression are `+` and `*`.

- The result of the expression is an a non-negative even integer.

- If the user enters an input, the program is guaranteed to terminate.

A property is only static if it is possible to compute it at compile-time. In general, most interesting program properties are uncomputable (e.g., does the program halt? is a particular variable guaranteed to be initialized?). There are two ways that uncomputability is handled in practice:

1. Make a conservative approximation to the desired property. E.g., for the halting problem answer either "yes, it halts" or "it may not halt".

2. Restrict the language to the point where it is possible to determine the property unequivocally. Such restrictions reduce the expressiveness of the language, but in return give precise static information. The ML language is an example of this approach; in order to provide static type information, it forbids many programs that would not give run-time type errors.

2 Types

Intuitively, types are sets of values. For instance, Java’s `int` type stands for the set of all integers (actually, the set of all integers representable using 32 bits), while the `boolean` type stands for the set of values `true` and `false`. In general, finer-grained distinctions might be helpful (e.g. even
integers, positive integers), but we will stick with the notion of disjoint types supported by most programming languages.

In Scheme (as well as all the toy languages we have studied thus far this semester), every value carries with it dynamic type information that is only checked when the value is examined during evaluation. For example:

- Evaluating a primitive application checks that the number and types of the operands are appropriate for the primitive operator.
- Evaluating an if expression checks that the test subexpression has boolean type.
- Evaluating a function application checks that the operator is a closure and that the number of actual arguments matches the number of formal parameters expected by the closure.

These sorts of run-time checks are the essence of **dynamic type checking**.

In most modern programming languages, the type of an expression is a static property, not a dynamic one. The choice between dynamic and static typing has been a source of a great debate in the programming language community. Adherents of static typing offer the following arguments in favor of static types:

- **Safety**: Static type checking eliminates certain kinds of errors that can occur at runtime. It is often extremely desirable to catch as many errors as possible before the program is run, especially in software that is safety critical (e.g., control software for airplanes, nuclear power plants, communication grids, and medical instruments) or financially important (e.g., software for banking, e-commerce sites, and satellites).
- **Efficiency**: Statically typed programs can execute more efficiently than dynamically typed ones, because no run-time storage is required for type information and static type checks eliminate the need to check types at run time.\(^1\)
- **Documentation**: Static types provide documentation about the program that can facilitate reasoning about the program, both by humans and by other programs (e.g., analyzers and translators). Such information is especially valuable in large programs. For example, programmers can often deduce how to use the operations of a data-structure library based on the names of the operations along with their argument types and return types.
- **Program Development**: Static types help programmers catch errors in their programs before running them and help programmers make representation changes. For example, suppose a programmer decides to change the interface of a procedure in a large program. The type checker helps the programmer by finding all the places in the program where there is a mismatch between the old and new interfaces.

Proponents of dynamic typing counter with the following claims:

- The restrictions placed on a language in order to make it statically type-checkable force the programmer into a straitjacket of reduced expressive power.
- In most statically typed languages, types serve mainly to make the language easier to implement, not easier for writing programs.

---

\(^1\) Even in statically typed languages, there are still run-time space and time overheads for representing and checking the tags of sum values, like lists and trees. Indeed, many dynamically typed languages can be viewed as having a single sum data type that describes all values.
• Finding type errors at analysis time is overrated. The hard-to-find errors that occur in practice are logical errors, not type errors. Finding logical errors requires testing programs with extensive test suites that would uncover type errors anyway. Eliminating the time-consuming static type-checking phase allows programmers to test for logical errors sooner in the program development process.

• Using programmer time more efficiently is more important than using computer resources more efficiently. Dynamically typed languages allow programmers to be more productive. They can build prototype systems more quickly because they don’t waste time wrestling with type annotations and static type checkers.

The relative benefits of static versus dynamic type checking are often debated in electronic forums. Some spirited discussions can be found at the Lambda The Ultimate programming languages weblog (http://lambda-the-ultimate.org/).

3 HOFLEMT: A Language with Monomorphic Types

In a language with monomorphic types, each expression can be assigned a single type. Here we consider monomorphic type systems in the context of the toy language HOFLEMT, a language that extends HOFL with Explicit Monomorphic Types. HOFLEMT is the first of a series of typed toy languages we will study. Just as the toy languages FOFL, FOBS, and HOFL gave us insight into interpretation and dynamically typed languages (like Scheme), the typed toy languages will give us insight into type checking, type reconstruction, and statically typed languages (like Java and OCaml).

3.1 HOFLEMT Syntax

Fig. 1 presents the grammar for HOFLEMT, a statically-typed version of HOFL with explicit monomorphic types. The grammar is similar to that for the dynamically-typed HOFL except for a few additions and changes.

The major addition is the introduction of type phrases via the non-terminals $B$ and $T$. According to the grammar, a type may be of three different forms:

1. **Base types** $B$ are names that designate the types of HOFL literals:
   - `unit` - the type of the one-point set \{()\};
   - `bool` - the type of the two-point set \{true,false\};
   - `int` - the type of integers;
   - `char` - the type of characters;
   - `string` - the type of strings;
   - `symb` - the type of symbols;

2. A list type of the form (listof $T$) designate lists all of whose elements have type $T$. In HOFLEMT, only **homogeneous** lists are supported – that is, lists in which all elements must be of the same type. For example (listof int) designates lists of integers, and (listof bool) designates lists of booleans, but it is not possible to have a list that contains both integers and booleans.

3. A **function type** of the form (\(\rightarrow (T_1 \ldots T_n) T_0\)) designates functions whose $n$ arguments, in order, have types $T_1 \ldots T_n$, and whose result has type $T_0$. For example, an
Figure 1: Grammar for the monomorphically typed \textsc{Hoflemt} language.
incrementing function on integers would have type \((\to \ (\text{int}) \ \text{int})\), an addition function on integers would have type \((\to \ (\text{int} \ \text{int}) \ \text{int})\), and a less-than function on integers would have type \((\to \ (\text{int} \ \text{int}) \ \text{bool})\).
In Hoflemt, the syntax of abstractions, recursions, and the empty list primitive application have been extended to include **type annotations**:

- In an abstraction \((\text{fun} \ ((I_1 \ T_1) \ldots \ (I_n \ T_n)) \ E)\), each formal parameter name is paired with the type of that parameter. For example, here is a function that takes an integer and a boolean; it increments the integer if the boolean is true, but doubles it if the boolean is false:

  \[
  \begin{align*}
  \text{(fun} \ ((n \ \text{int}) \ (b \ \text{bool})) \\
  \text{(if} \ b \ (+ \ n \ 1) \ (* \ n \ 2)))
  \end{align*}
  \]

  As another example, consider a function that composes a string to integer function with an integer to boolean function:

  \[
  \begin{align*}
  \text{(fun} \ ((f \ (-> \ \text{int} \ \text{bool})) \ (g \ (-> \ \text{string} \ \text{int}))) \\
  \text{(fun} \ ((x \ \text{string})) \\
  \text{(f} \ (g x)))
  \end{align*}
  \]

- In a local recursion \((\text{bindrec} \ ((I_1 \ T_1 \ E_1) \ldots \ (I_n \ T_n \ E_n)) \ E)\), each binding is annotated with the type of that binding. For example:

  \[
  \begin{align*}
  \text{(hoflemt} \ (n) \\
  \text{(bindrec} \ ((\text{even?} \ (-> \ \text{int} \ \text{bool})) \ (\text{fun} \ ((n \ \text{int})) \\
  \text{(if} \ (= \ n \ 0) \ #t \\
  \text{(odd?} \ (- \ n \ 1)))]) \\
  \text{(odd?} \ (-> \ \text{int} \ \text{bool})) \\
  \text{(fun} \ ((n \ \text{int})) \\
  \text{(if} \ (= \ n \ 0) \ #f \\
  \text{(even?} \ (- \ n \ 1))))]) \\
  \text{(even?} \ n))
  \end{align*}
  \]

- The empty list primitive application has a type annotation that indicates what type of empty list is being created. For example:

  - \((\text{empty} \ \text{int})\) creates an empty list of integers;
  - \((\text{empty} \ (-> \ \text{int} \ \text{bool}))\) creates an empty list of integer predicates; and
  - \((\text{empty} \ (\text{listof} \ \text{int}))\) creates an empty list of integer lists.

As in HOFL all program arguments are assumed to be integers, so the formal parameters of programs do not require any type annotations.

The kernel syntax of Hoflemt differs from HOFL in a few other ways:

- In Hoflemt, multiple-argument abstractions \((\text{fun})\) and multiple-argument applications are kernel forms rather than syntactic sugar. The reason for this is that Hoflemt does not support currying so that its type system can track the number of arguments of a function.

- In Hoflemt, bindpar is a kernel form. Unlike in HOFL, it cannot be expressed as an application of a fun because the types of the function arguments are not apparent. For example, the bindpar expression

  \[
  \begin{align*}
  \text{(bindpar} \ ((i \ (+ \ x \ y)) \ (b \ (< \ x \ y))) \\
  \text{(if} \ b \ i \ (* \ 2 \ i)))
  \end{align*}
  \]

  cannot be written in the form
(hoflemt (I_{formal_1} \ldots) \ E_{body} \ (\text{def} \ I \ T \ E_1 \ \ldots)) \sim (hoflemt (I_{formal_1} \ldots) \ (\text{bindrec} ((I_1 \ T_1 \ E_1) \ \ldots \ \ E_{body})))

(\text{def} (I_{fcn} (I_1 \ T_1) \ \ldots) \ E_{body}) \sim (\text{def} I_{fcn} (\text{fun} ((I_1 \ T_1) \ \ldots) \ E_{body}))

(\text{bind} I_{name} \ E_{defn} \ E_{body} \sim (\text{bindpar} ((I_{name} \ E_{defn}) \ E_{body}))

(\text{bindseq} (((I) \ E) \ \ldots) \ E_{body}) \sim (\text{bind} I \ E \ (\text{bindseq} (\ldots) \ E_{body}))

(\text{bindseq} () \ E_{body}) \sim E_{body}

(\&\& E_{rand_1} \ E_{rand_2}) \sim (\text{if} E_{rand_1} \ #f)

(|| E_{rand_1} \ E_{rand_2}) \sim (\text{if} E_{rand_1} \ #t \ E_{rand_2})

(\text{cond} \ (\text{else} \ E_{default})) \sim E_{default}

(\text{cond} (E_{test} \ E_{default}) \ \ldots) \sim (\text{if} E_{test} \ E_{default} \ (\text{cond} \ \ldots))

(list T) \sim (\text{empty} T)

(list T \ E_{hd} \ \ldots) \sim (\text{prep} E_{hd} \ (\text{list} T \ \ldots))

\text{Figure 2: Desugaring rules for Hoflemt.}

\text{without knowing the types} \ T_i \ \text{and} \ T_b, \ \text{which aren’t explicit.}

The syntactic sugar of \textsc{Hoflemt} (Fig. 2) is similar to that of \textsc{Hofl}. There are a few notable differences:

- The sugar \textbf{bind} expression desugars to a one-argument kernel \textbf{bindpar} expression. (As in \textsc{Hofl}, \textbf{bindseq} is still sugar for a sequence of nested \textbf{bind} expressions.)

- The \textbf{list} sugar (\textbf{list} \ T \ E_1 \ \ldots \ E_n) requires an explicit type \textit{T} for the elements of the list. Without this type, the base case of the \textbf{list} desugaring (into (\textbf{empty} \ T)) couldn’t be performed.

- The definitions of sugared \textsc{Hoflemt} programs require addtional type annotations for desugaring into \textbf{bindrec} expressions that may involve \textbf{fun} expressions.

\textbf{3.2 Examples}

The \textsc{Hoflemt} program in Fig. 3 illustrates all three different kinds of type annotations. The type annotations have been highlighted in slanted font for emphasis. Make sure you can justify to yourself why all the type annotations are the way they are.

As we shall see below, the explicit type annotations in \textsc{Hoflemt} are designed to support automatic type checking. It turns out that the \textsc{Hoflemt} annotations are the minimal set of annotations that allow the expression to be type checked via a simple type ”evaluator” that ”evaluates” each expression to its type. Program parameters do not need to be annotated since they are assumed to be integers. Unlike \textbf{bindrec}, the \textbf{bind} and \textbf{bindpar} constructs do not require the type of the named definition(s) to be given an explicit type. This is because in these constructs the, the type checker can determine the type of the name(s) from the type of the definition(s). In contrast, the
Figure 3: Example of Hoflemt program.

The recursive scope of \texttt{bindrec} makes it generally necessary to know the type of a recursively bound name(s) as part of calculating the type of the associated definition(s).

The type system of \texttt{Hoflemt} is \textbf{monomorphic}, which means that every \texttt{Hoflemt} expression has exactly one type. When trying to write \texttt{Hofl} expressions in \texttt{Hoflemt}, monomorphism can require duplicating a function that is used at different types. For example, consider the \texttt{Hofl} expression:

\begin{verbatim}
(bindpar ((app5 (fun (f) (f 5)))
   (make-sub (fun (n x) (- x n)))
   (app5 (make-sub ((app5 make-sub) 3))))
\end{verbatim}

The corresponding expression in \texttt{Hoflemt} is:

\begin{verbatim}
(bindpar ((app5\_1 (fun ((f -> (int) int))) (f 5))
   (app5\_2 (fun ((f -> (int) (-> (int) int))) (f 5))
   (make-sub (fun ((n int)) (fun ((x int)) (- x n))))))
\end{verbatim}

There are two copies of \texttt{app5} because the function argument of \texttt{app5} has a different type in the two applications:

- In \texttt{app5\_1}, the argument \texttt{f} has type \((\rightarrow \text{(int)} \text{int})\);
- In \texttt{app5\_2}, the argument \texttt{f} has type \((\rightarrow \text{(int)} \rightarrow \text{(int)} \text{int}))\).

Monomorphism can be frustrating because it requires the programmer to duplicate code in order to satisfy the type system. You have experienced such code duplication first-hand when manipulating lists in \texttt{Java CS111}. If \texttt{Java}'s \textbf{generic types} (which aren't introduced until CS230) aren't used, then implementing a list of integers and a list of strings requires two different classes because the element types of the lists differ.

Examples of real-life monomorphic languages include C, Pascal, and Fortran. As suggested by the above example, in monomorphic languages it may be necessary to create many copies of the
same function that differ only in their type. For example, in monomorphic languages, it is necessary to write separate sorting routines for arrays of integers and arrays of floating point numbers because these two arrays have different types! Even worse, in Pascal, the size of the array is part of the array type, so one must write a different sorting function to sort arrays of 10 integers and arrays of 11 integers!

4 Type Checking

4.1 Well-Typedness

A Hoflemt expression $E$ is said to be well-typed if it is possible to prove that it has a type $T$ using a set of typing rules.

If $E$ is a closed expression, we use the notation $E : T$ to indicate that $E$ is a well-typed expression with type $T$. For example:

\[
\begin{align*}
() &: \text{unit} \\
\text{true} &: \text{bool} \\
5 &: \text{int} \\
"\text{foo}" &: \text{string} \\
\text{(symbol cs251)} &: \text{sym} \\
\text{(prepend 42 (prepend -17 \text{(empty int))})} &: \text{(listof int)} \\
\text{(fun ((a int) (b int)) (div (+ a b) 2))} &: \text{(- (int int) bool)}
\end{align*}
\]

To handle open expressions (expressions with free variables), we need the analog of value environments from the environment model of evaluation. Type environments are environments that associate value variable names with types. We will write type environments as sets of bindings of the form $E : T$. For example, the type environment

\[
\{a : \text{int}, b : \text{bool}, f : \text{(- (int int) bool)}\}
\]

associates the name $a$ with the type $\text{int}$, the name $b$ with the type $\text{bool}$, and the name $f$ with the type $\text{(- (int int) bool)}$. The primitive type environment $T_{E_{prim}}$ describes the types of Hoflemt’s primitive operators.

\[
T_{E_{prim}} = \{\not : \text{(- (bool bool) bool)}, \quad \text{and} : \text{(- (bool bool) bool)}, \\
\or : \text{(- (bool bool) bool)}, \quad \text{bool=} : \text{(- (bool bool) bool)}, \\
\leq : \text{(- (int int) bool)}, \quad \lt : \text{(- (int int) bool)}, \\
\eq : \text{(- (int int) bool)}, \quad \neq : \text{(- (int int) bool)}, \\
\geq : \text{(- (int int) bool)}, \quad \gt : \text{(- (int int) bool)}, \\
\ast : \text{(- (int int) int)}, \quad \div : \text{(- (int int) int)}, \\
\% : \text{(- (int int) int)}, \quad \text{sym=} : \text{(- (symb symb) bool)} \\
\ldots\}
\]

Figure 4: Primitive type environment $T_{E_{prim}}$ for Hoflemt.

If $TE$ is a type environment, $I$ is an identifier, and $T$ is a type, we use the notation $TE(I)$ to denote the type bound to $I$ in type environment $TE$, and

\[
TE + \{I_1 : T_1, \ldots, I_n : T_n\}
\]

to stand for the environment $TE$ extended with bindings between $I_1 \ldots I_n$ and $T_1 \ldots T_n$, respectively.
4.2 Proving Expressions Well-Typed

Just as expressions can be evaluated relative to a value environment, expressions can be typed relative to a type environment. We now describe a formal process for determining the types of Hoflemt expressions relative to a type environment. The assertion that an expression $E$ has type $T$ with respect to type environment $TE$ is known as a type judgment and is written

$$TE ⊢ E : T$$

This is pronounced “expression $E$ has type $T$ in type environment $TE$” or, more loosely, “$TE$ proves that $E$ has type $T$.” When such an assertion is true, we say that the type judgment is valid. If $TE ⊢ E : T$ is valid for some $T$, we say that $E$ is well typed with respect to $TE$. Otherwise, $E$ is ill typed with respect to $TE$. If the type environment is understood from context, we just say that $E$ is well typed or ill typed.

Valid type judgments can be determined via type rules. Each type rule has the form

$$\frac{\text{premise}_1 \ldots \text{premise}_n}{\text{conclusion}} \quad \text{[rule-name]}$$

where conclusion and each premise_i are type judgments, and [rule-name] is the name of the rule. A rule without any premises is an axiom, and is written without a horizontal line. We will call a type rule with a nonempty set of premises an inference rule. If all of the premises of a rule are valid, then the type judgment in the conclusion of the rule is valid.

The type rules for Hoflemt are presented in Fig. ?? (we will flesh these out in class). The [unit], [bool], [int], [char], [string], [symb], and [error] rules are axioms that hold for any type environment $TE$. Type rules are really rule schemas in which every domain variable can be instantiated by any element of its domain. So [int] represents the infinite number of rules that can be obtained by instantiating $TE$ with any type environment and $N$ with any integer literal.

The [if] rule requires that the test expression denote a boolean and the two branches have the same type $T$. If these requirements are met, the type of the if expression is the type $T$ of the branches. The constraint that the two branch types and return type must all be the same is specified by using the same domain variable, $T$, for all three types.

Many of the ways to instantiate the [if] rule schema may not make sense at first glance. For example, here is one instantiation of the if rule:

$$\{\} ⊢ 1 : \text{bool} \quad \{\} ⊢ 2 : \text{symb} \quad \{\} ⊢ 3 : \text{symb}$$

$$\{\} ⊢ (\text{if } 1 \ 2 \ 3) : \text{symb}$$

Certainly we should not be able to prove that (if 1 2 3) has type symb! But the rule doesn’t say that (if 1 2 3) has type symb. Rather, it says that (if 1 2 3) would have type symb if the integer 1 had type bool and the integers 2 and 3 had type symb. But it is impossible to prove these false premises, and so the false conclusion will never be declared to be a valid judgment by the type system.

The type rules in Fig. ?? can be used to prove that a given Hoflemt expression is well-typed. A proof that expression $E$ is well-typed with respect to a type environment $TE$ is a tree of type judgements where:

- The root of the tree is $TE ⊢ E : T$ for some type $T$;

- Each judgement $J$ appearing in the tree is justified by instantiating one of the typing rules such that $J$ is the conclusion of the instantiated rule and the children judgements of $J$ are the hypotheses of the instantiated rule.

Such a tree of judgements whose root is the judgement $J$ is said to be a type derivation (or typing) for $J$. 
Figure 5: Type rules for the HOFLEMT language.
For example, consider the expression

\[
\text{(bindpar ((app5 (fun ((f -> (int) bool))) (f 5)))}
\text{(app5 (fun ((x int)))}
\text{ (> x 0)))}
\]

Suppose that we want to show that this expression is well-typed with respect to the empty environment. Because the typing derivation will be a rather wide tree, we will introduce the following abbreviations to make it narrower:

- \( \text{TIB} = (\rightarrow (\text{int}) \text{bool}) \)
- \( \text{TIBB} = (\rightarrow (\text{TIB}) \text{bool}) \)
- \( \text{Eabsf} = (\text{fun} ((f \text{TIB})) (f 5)) \)
- \( \text{Eabsx} = (\text{fun} ((x \text{int})) (> x 0)) \)
- \( \text{Ebind} = (\text{bindpar} ((\text{app5 Eabsf})) (\text{app5 Eabsx})) \)
- \( \text{TE1} = \{f: \text{TIB}\} \)
- \( \text{TE2} = \{\text{app5: TIBB}\} \)
- \( \text{TE3} = \{\text{app5: TIBB, x:int}\} \)

Below, we will flesh out a type derivation for the expression that proves that it has type bool. Each horizontal line is labeled with the name of the instantiated rule. Note that the leaves of the typing derivation are judgements involving literals or variables; these have no hypotheses. Also note that the "shape" of the derivation is an "upside down" abstract syntax tree for the expression at the root. That is, a judgement for an expression \( E \) follows from the judgements of its direct subexpressions.

As shown above, type derivations can be drawn as trees in which all hypotheses for a rule are on the same line above the horizontal bar and the conclusion of a rule is below the horizontal bar. We shall call this the \textbf{horizontal format} for a type derivation.
Using the horizontal format, it is very easy to run out of horizontal space when drawing a type derivation. Below, we illustrate an alternative **vertical format** for displaying the above type derivation that makes much better use of horizontal space:

```
+ (var) TE1 |- f : TIB
+ (int) TE1 |- 5 : int
+ (app) TE1 |- (f 5) : bool
+ (fun) |- (fun ((f TIB)) (f 5)): TIBB
  | + (var) TE2 |- app5 : TIBB
  | | + (var) TE3 |- x : int
  | | + (int) TE3 |- 0 : int
  | | + (gt) TE3 |- (> x 0) : bool
  | + (fun) TE2 |- (fun ((x int)) (> x 0)) : TIB
+ (app) TE2 |- (app5 Eabsx) : bool
(bindpar) |- (bindpar ((app5 Eabsf)) (app5 Eabsx)) : bool
```

In this alternative representation, each conclusion of a rule is labeled with the name of the rule used to derive it, and the hypotheses of the rule are those judgements on the lines labelled "+" directly above the leftmost character of the rule name. Vertical lines are used to connect the hypotheses of the same rule.

The vertical format makes it easier to draw type derivations for more complex expressions using fewer abbreviations without running out of space. For example, Fig. ?? shows a type derivation for the following expression:

```
(bindpar ((app5_1 (fun ((f -> (int) int))) (f 5))
   (app5_2 (fun ((f -> (int) -> (int) int)))) (f 5))
   (make-sub (fun ((n int)) (fun ((x int)) (- x n))))))
   (app5_1 (make-sub ((app5_2 make-sub) 3)))
```

The type derivation uses the following abbreviations:

```
TII  = (-> (int) int)
TE1  = {app5_1: (-> (TII) int),
         app5_2: (-> ((-> (int) TII) TII),
         make-sub: (-> (int) TII)}
```

As mentioned earlier, the above derivation contains two separate copies of the `app5` function due to the monomorphic nature of HOFLEMT.

Above we only considered showing that HOFLEMT expressions are well-typed. It is also possible to show that HOFLEMT programs are well-typed. This can be done by showing that the body of the program is well-typed with respect to a type environment where each program parameter is bound to the int type.

### 4.3 Type Checking

It is possible to check the well-typedness of a HOFLEMT expression or program via an automatic **type checker**. A type checker is very much like an evaluator, except that rather than finding the type of an expression relative to a value environment, it determines the type of an expression relative to a type environment.

### 4.4 Type Soundness

Types are important in HOFLEMT because a well-typed program cannot encounter certain kinds of errors when executed. This is a consequence of the following type soundness theorem:
Figure 6: Example type derivation using the vertical format.

Theorem 4.1 (Type Soundness). For any well-typed Hoflemt expression \( E \) that has a type \( T \), if \( E \) evaluates to a value \( V \) at run-time, then \( V \) is guaranteed to be a member of the set of values denoted by \( T \). 

The type soundness theorem means that it is impossible to encounter dynamic type-checking errors when evaluating a well-typed expression at run-time.

Note that the theorem contains the conditional "if \( E \) evaluates to a value \( V \)”. It is possible that the evaluation of \( E \) may signal an error or loop infinitely. In such cases, a value \( V \) is never produced, and the theorem is trivially true. Thus, an expression that necessarily signals an error or loops can be assigned any type whatsoever, and the theorem still holds.

The type soundness theorem is often summed up by the motto “Well-typed programs do not go wrong”. This motto is somewhat deceptive – well-typed programs can encounter errors at run-time, but those errors cannot be type errors. Other errors that can still be encountered are errors that depend on particular values (e.g. divide-by-zero, attempt to take the head of an empty list, accessing an array at an out-of-bounds index) as well as logical errors in the program (it gives the wrong answer).

Type soundness is important for two main reasons:

- The programmer and user can be confident that certain kinds of errors cannot occur when the program is executed at run-time. This is important for software in safety-critical systems where run-time errors cannot be tolerated.
• The execution of the program can be more efficient. Because dynamic type errors cannot occur, run-time values do not have to be tagged with their types (saving space) and these tags do not have to be checked at run time (saving time).

5 Type Reconstruction: The Idea

In Hoflmt, it is necessary to specify explicit type information in certain situations in order to guide type-checking. For example:

• All formal variables declared by `fun` 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.

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:

```
(fun (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 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 the `fun` expression studied above. Suppose that:

- `?fun` is the type of the result of evaluating the `fun` expression;
- `?if` is the type of the result of evaluating the `if` expression;
- `?f` is the type of `f`;
- `?g` is the type of `g`;
- `?x` is the type of `x`;
- `?y` is the type of `y`;

Here, subscripted versions of `?` 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:

\[
\begin{align*}
?fun &= (\to (?f ?g ?x ?y) ?if) & \text{abs has a function type from args to body} \\
?f &= (\to (\text{int} ?y) ?res1) & \text{In first call to } f, \text{ rator has type from arguments to result} \\
?f &= (\to (?x \text{ string}) ?res2) & \text{In second call to } f, \text{ rator has type from arguments to result} \\
?g &= (\to (?x) ?res3) & \text{In call to } g, \text{ rator has type from arguments to result} \\
?res1 &= \text{bool} & \text{Type of if test must be bool} \\
?res2 &= ?res3 & \text{if branches must have same type} \\
?if &= ?res2 & \text{if has type of first branch}
\end{align*}
\]

Above, the types `?res1`, `?res2`, and `?res3` have been introduced as the names of components of `?f` and `?g`.

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

\[
\begin{align*}
?x &\mapsto \text{int} \\
?y &\mapsto \text{string} \\
?if &\mapsto ?res1 = ?res2 = ?res3 = \text{bool} \\
?f &\mapsto (\to (\text{int string}) \text{ bool}) \\
?g &\mapsto (\to (\text{int}) \text{ bool}) \\
?abs &\mapsto (\to ((\to (\text{int string}) \text{ bool}) (\to (\text{int} \text{ bool}) \text{ int string}) \text{ bool})
\end{align*}
\]

It turns out that equations involving type variables can be automatically solved by a process known as unification. (Sadly, we won’t have time to study unification this semester.) The above reasoning process can be automated by modifying a type checker to collect a set of equations involving type variables and using unification to solve them.

Note that a collection of type equations need not always have the neat form of solution indicated by the example. For example, suppose we modify the above example to be:

\[
\begin{align*}
&\text{abs (f g x y)} \\
&(\text{if (f 3 4) (f x "static") (g x)})
\end{align*}
\]

The collection of equations for the modified example 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 collection of type equations may be underconstrained, as in the following perturbation of the above example:
In this case, the type of \( x \) is unknown, and the type deduced for the expression is

\[
\texttt{(fun (f g x y) (if (f x y) (f x "static") (g x)))}
\]

\[
\text{(-} \rightarrow \texttt{(-> (?x string) bool) (-} \rightarrow \texttt{(?x bool) ?x bool) bool})
\]

The appearance of \(?x\) in this type implies that \(?x\) can be instantiated with any type. That is, the type of the abstraction is polymorphic in \(?x\). In a language with so-called universal polymorphism, this polymorphic type would be expressed as

\[
\texttt{(forall (t) (-} \rightarrow \texttt{((-} \rightarrow \texttt{(t string) bool) (-} \rightarrow \texttt{(t bool) t string) bool}))}
\]