Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 8: May 1, 2012

Lecturer: Mooly Sagiv *Scribe: Dvir Netanely and Guy Rozendorn*

8.1 Fundamentals

Programming involves a wide range of computational constructs, such as data structures, functions, objects, communication channels, and threads of control.

Because programming languages are designed to help programmers organize computational constructs and use them correctly, many programming languages organize data and computations into collections called *types*.

A *type system*, specified for each programming language, controls the ways typed programs may behave, and makes behavior outside these rules illegal.

There are three main uses of types in programming languages.

8.1.1 Abstraction

Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can think of a string as a collection of character values instead of as a mere array of bytes.

8.1.2 Documentation

Types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be represented as integers, but if a programmer declares a function as returning a timestamp type rather than merely an integer type, this documents part of the meaning of the function.

8.1.3 Optimization

Types can be used to provide information to the compiler about data manipulated by the program. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions.

8.1.4 Safety

Use of types may allow a compiler to detect meaningless or probably invalid code. For example, we can identify an expression: 4 / "Hi" as invalid, because the rules of arithmetic do not specify how to divide an integer by a string.

The following table characterizes the type safety of some common programming languages. We will discuss each form of type error listed in the table in turn.

Safety	Example languages	Explanation
Not safe	C;C++	Type casts; pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	ML; Haskell; Java	Complete type checking

Table 8.1: Type safety of some common programming languages

Type Casts. Type casts allow a value of one type to be used as another type. For example, In C, an integer can be cast to a function, allowing a jump to a location that does not contain the correct form of instructions to be a C function.

Pointer Arithmetic. The expression *(p+i) has type A if p is defined to have type A*. Because the value stored in location p+i might have any type, an assignment like x = *(p+i) may store a value of one type into a variable of another type and therefore may cause a type error.

Explicit Deallocation and Dangling Pointers. In Pascal, C, and some other languages, the location reached through a pointer may be deallocated (freed) by the programmer. This creates a dangling pointer, a pointer that points to a location that is not allocated to the program. If p is a pointer to an integer, for example, then after we deallocate the memory referenced by p, the program can allocate new memory to store another type of value. This new memory may be reachable through the old pointer p, as the storage allocation algorithm may reuse space that has been freed. The old pointer p allows us to treat the new memory as an integer value, as p still has type int. This violates type safety.

8.2 What is a type

A program typically associates each value with one particular type. Other entities, such as objects, modules, communication channels, dependencies, or even types themselves, can become associated with a type.

Some implementations might make the following identifications:

- class; a type of an object
- data type; a type of a value
- kind; a type of a type

8.2.1 Primitive data types

A primitive data type is either of the following:

- a basic type is a data type provided a basic building block. Most languages allow composite types to be recursively constructed starting from basic types.
- a built-in type is a data type for which the programming language provides built-in support.

In most programming languages, all basic data types are built-in.
Classic basic primitive types may include:

- Character (character, char);
- Integer (integer, int, short, long, byte) with a variety of precisions;
- Floating-point number (float, double, real, double precision);
- Boolean, logical values true and false.
- Reference (also called a pointer or handle), a small value referring to another object's address in memory, possibly a much larger one.

8.2.2 Composite data type

A composite data type is any data type which can be constructed in a program using its programming language's primitive data types and other composite types. The act of constructing a composite type is known as composition. Several examples of composite data types:

- list
- structures

8.3 Type errors

A type error is erroneous or undesirable program behaviour caused by a discrepancy between differing data types. For example, if an integer value is used as a function, this is a type error. A common type error is to apply an operation to an operand of the wrong type. For example, it is a type error to use integer addition to add a string to an integer. Dividing an Integer by zero is a runtime error, not a type error.

8.4 Type checking

The process of verifying and enforcing the constraints of types (a.ka. type checking) may occur either atcompile-time (a static check) or run-time (a dynamic check), and it is used to prevent some or all type errors.

8.4.1 Static typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time, and the developr is warned about the error before the program is given to other users or shipped as a product.

More specifically, most checkers are both sound and conservative. A type checker is sound if no programs with errors are considered correct. A type checker is conservative if some programs without errors are still considered to have errors.

Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed.

Program execution may also be made more efficient (e.g. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations. Because they evaluate type information during compilation and therefore lack type information that is only available at run-time, static type checkers are conservative. They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed.

For example, even if an expression `<complex test>` always evaluates to true at run-time, a program containing the code:

Example 1

```
if <complex test> then 42 else <type error>
```

will be rejected as ill-typed, because a static analysis cannot determine that the else branch won't be taken. Static typing helps by providing strong guarantees of a particular subset of commonly-made errors never occurring.

8.4.2 Dynaming typing

In programming languages with run-time type checking, the compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct type. For example, the Lisp language operation car returns the first element of a cons cell. Because it is a type error to apply car to something that is not a cons cell, Lisp programs are implemented so that, before (car x) is evaluated, a check is made to make sure that x is a cons cell. An advantage of run-time type checking is that it catches type errors.

Development in dynamically typed languages is often supported by programming practices such as unit testing. In practice, the testing done to ensure correct program operation can detect a much wider range of errors than static type-checking, but full test coverage over all possible executions of a program (including timing, user inputs, etc.), if even possible, would be extremely costly and impractical.

8.4.3 Combinations of dynamic and static typing

The presence of static typing in a programming language does not necessarily imply the absence of all dynamic typing mechanisms. For example, Java supports downcasting and other type operations that depend on run-time type checks, a form of dynamic typing.

More generally, most programming languages include mechanisms for dispatching over different 'kinds' of data, such as disjoint unions, polymorphic objects, and variant types: Even when not interacting with type annotations or type checking, such mechanisms are materially similar to dynamic typing implementations.

Certain languages, for example Cython (C extensions for Python), are dynamically typed by default, but allow this behaviour to be overridden through the use of explicit type hints that result in static typing. One reason to use such hints would be to achieve the performance benefits of static typing in performance-sensitive parts of code.

8.4.4 Strongly-typed languages and Weakly-typed languages

A type system is said to feature strong typing when it specifies one or more restrictions on how operations involving values of different data types can be intermixed. A computer language that implements strong typing will prevent the successful execution of an operation on arguments that have the wrong type.

Weak typing means that a language implicitly converts (or casts) types when used. Consider the following example:

Example 2

```
var x := 5;    // (1)  (x is an integer)
var y := "37"; // (2)  (y is a string)
x + y;         // (3)  (?)
```

In a weakly typed language, the result of this operation depends on language-specific rules. Visual Basic would convert the string "37" into the number 37, perform addition, and produce the number 42. JavaScript would convert the number 5 to the string "5," perform string concatenation, and produce the string "537." In JavaScript, the conversion to string is applied regardless of the order of the operands (for example, y + x would be "375") while in AppleScript, the left-most operand determines the type of the result, so that x + y is the number 42 but y + x is the string "375".

Weakly typed programming languages are those that support either implicit type conversion (nearly all languages support at least one implicit type conversion), ad-hoc polymorphism (also known as overloading) or both. These less restrictive usage rules can give the impression that strict adherence to typing rules is less important than in strongly typed languages and hence that the type system is "weaker".

8.4.5 Static and dynamic type checking in practice

The choice between static and dynamic typing requires trade-offs:

Static typing can find type errors reliably at compile time. This should increase the reliability of the delivered program.

Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce optimized machine code. Further, compilers for statically typed languages can find assembler shortcuts more easily.

By contrast, dynamic typing may allow compilers to run more quickly and allow interpreters to dynamically load new code, since changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit. This may reduce the edit-compile-test-debug cycle.

Dynamic typing allows constructs that some static type checking would reject as illegal. For example, eval functions, which execute arbitrary data as code, become possible. An eval function is possible with static typing, but requires advanced uses of algebraic data types. Furthermore, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full-fledged data structure (usually for the purposes of experimentation and testing).

Dynamic typing typically makes metaprogramming more effective and easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code. More advanced run-time constructs such as metaclasses and introspection are often more difficult to use in statically typed languages. In some languages, such features

may also be used e.g. to generate new types and behaviors on the fly, based on run-time data. Such advanced constructs are often provided by dynamic programming languages; many of these are dynamically typed, although dynamic typing need not be related to dynamic programming languages.

8.5 Explicit type declaration and inference

Statically typed languages (such as C and Java) require that programmers declare the types they intend a method or function to use. This can serve as additional documentation for the program, which the compiler will not permit the programmer to ignore or permit to drift out of synchronization. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala): the compiler draws conclusions about the types of variables based on how programmers use those variables. Explicit type declaration is not a necessary requirement for static typing in all languages.

For example, given a function f(x, y) that adds x and y together, the compiler can infer that x and y must be numbers since addition is only defined for numbers. Therefore, any call to f elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.

Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression 3.14 might imply a type of floating-point, while [1, 2, 3] might imply a list of integers typically an array.

8.6 Summary of popular languages

C
: static; explicit; not safe

Java
: static; explicit; safe

Haskell
: static; implicit; safe

Python
: dynamic; implicit; safe

JavaScript
: dynamic; implicit; not safe

8.7 Type Inference

Type inference refers to the process of determining the type of expressions based on known types of some symbols that appear in them. Using a type inference algorithm, the compiler is often able to infer the type of a variable or the type signature of a function without explicit type annotations having been given by the programmer.

Whereas in type-checking, the type of the identifiers in known and the algorithm only needs to verify that it is being used according to type constraints, type-inference algorithms need to learn the types of expressions using logical inference from the way they are used.

8.7.1 Simple Type-Inference Examples

In order to illustrate the basic idea of type-inference we shall use a special simplified version of the Haskell programming language, named μHaskell. In μHaskell, all constants, built-in operators, and other functions have a single predefined type which makes it easier to deduce the type of an expression based on its usage in the program. A type in μHaskel is defined to be either a single type identifier (”constant” type), a type parameter (type ”variable”), or a function type mapping one type to one other type. Multiple-parameter functions are dealt with by currying, that is, X -¿ Y -¿ Z.

Example 3

```
f1 x = x + 2
f1 :: Int -> Int
```

In this example, the type of the function f1 is inferred to be `Int->Int` from the fact the 2 is Int and the + operator is of type `Int->Int`.

Example 4

```
f2 (g,h) = g(h(0))
f2 :: (a -> b, Int -> a) -> b
```

In this example, we infer that h is a function of type `Int->a` (a is a type variable denoting unknown type). We further infer that a is the argument type of function g, which returns a second unknown type denoted as b. Therefore h is marked as type `Int->a` and g is marked as type `a->b`. The return type of function f2 is equal to the return type of function g and there for it is b.

8.7.2 The Hindley-Milner Type-inference algorithm

The type-inference algorithm described below was invented by Robin-Milner for the ML programming language and was later implemented similarly in several other programming languages. Similar ideas were independently developed by Curry and Hindley in connection with the study of lambda calculus, and so the algorithm is currently known as the HindleyMilner algorithm for type inference.

The algorithm computes a constraint set based on how each value is used in the given expression and builds a parse tree that represents all sub-expressions and their constraints. It then uses a technique called "Unification" to infer the most general type for the given expression parse tree.

Properties of the Hindley-Milner algorithm

- Completeness - It finds a solution if one exists; otherwise, it correctly reports that no solution is possible.

- Fast - can compute a type almost in linear time with respect to the size of the source, making it practically usable to type large programs

- It can deduce the most general type of a given source without the need of any type annotations or other hints from the programmer.

Algorithm steps

1. Build a parse tree starting with the given expression and including all its sub expressions.

2. Assign a type to the expression and each sub-expression.

   - For any compound expression or variable, use a type variable.
   - For known operations or constants, such as + or 3, use the type that is known for this symbol.

3. Generate a set of constraints on types, using the parse tree of the expression. These constraints reflect the fact that if a function is applied to an argument, for example, then the type of the argument must equal the type of the domain of the function.

4. Use the "Unification" technique to solve the constraints.

More about the Unification method

Unification is an algorithmic process by which one attempts to solve a system of equations based on a series of term-substitutions. In each step, the goal of unification is to find a substitution which demonstrates that two seemingly different terms are in fact identical.

Given two input terms s and t, unification is the process which attempts to find a substitution that structurally identifies s and t. If such a substitution exists, we call this substitution a unifier of s and t. In theory, some pairs of input terms may have infinitely many unifiers. However, for most applications of unification, it is sufficient to consider a most general unifier (MGU) as all other unifiers are its instances.

As mentioned above, the Hindley-Milner is using Unification in its last step to infer the most general type for the examined expression. After defining a set of constraints on types and type-variables representing the type usage in the program, the algorithm executes a series of constraint pair unifications until reaching a solution for the constraint set.

8.7.3 Execution examples for the Hindley-Milner algorithm

Example 5 We start by running the algorithm on example 3 above:

```
f1 x = x + 2
f1 :: Int -> Int
```

Building the parse tree. In this step we build the parse tree:

- Start with a root named "Fun" to indicate this is a parse tree of a function declaration.
- Add the actual function name and the function arguments as root children (in this case "add" and "x")
- Add a third root child as the parse-tree of the function body
- Nodes labeled "@" denote function application where the left child is applied to the right child
- The operator '+' is treated as a curried function (such as (+) x 2 in Haskell)
- Constants such as operators and numbers get their own nodes
- Variables also get their own nodes, and we link them back to their binding occurrence using a dashed line.

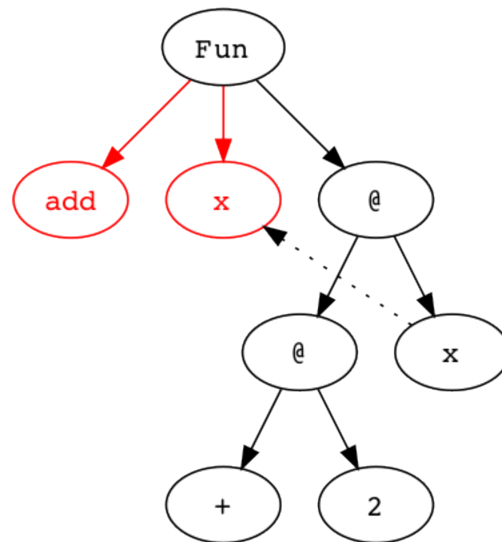

Figure 8.1: Parse tree for add function

Assigning a type variable to the expression and each sub-expression. In this step we redraw the graph, adding a type variable next to each node (named `t_0, t_1, ...`). Nodes holding the same argument will be assigned with the same type-variable (see `t_1` in figure 2 below).

Generating a set of constraints. In this step we use the parse tree of the expression to define relations between the type variables. The constraint generated at each node depends upon the node type. For constant expressions, we assign the type variable with the known type for the constant.

- For the constant 2, we thus add the constraint `t_3 = Int`.
- For the operator + which is known to be of type `Int -> (Int -> Int)` we add the constraint `t_2 = Int -> (Int -> Int)`.

Variable do not supply information about the way they are used and so no constraints are added for variable nodes.

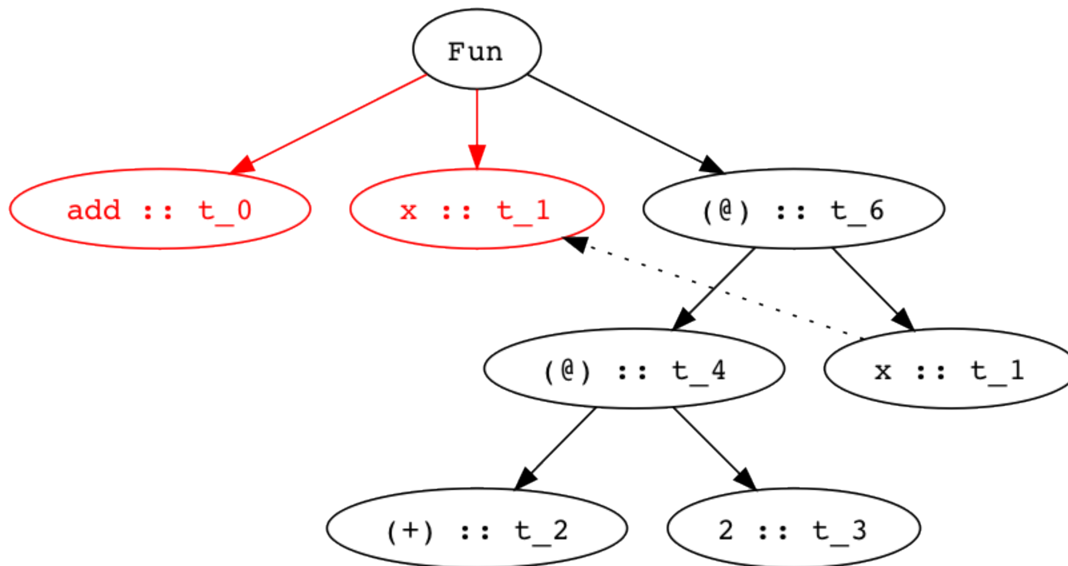

Figure 8.2: Parse tree labeled with type variables

For function application nodes (@ nodes), If expression f is applied to expression a, then f must have a function type. In addition, the type of a must be the type of the domain of this function, and the type of f a must be the type of the result of the function. In symbols, if the type of f is `t_f`, the type of a is `t_a`, and the type of f a is `t_r`, then we must have `t_f = t_a -> t_r`.

- For the sub-expression `@ (+) 2` we add the constraint `t_2 = t_3 -> t_4`.
- For the sub-expression `@ (@ (+) 2) x` we add the constraint `t_4 = t_1 -> t_6`.
- For the root node, which is the function definition node, the constraint is a type from the type of the argument to the type of the body.
- For the sub-expression `add x = @ (@ (+) 2) x`we add the constraint `t_0 = t_1 -> t_6`.

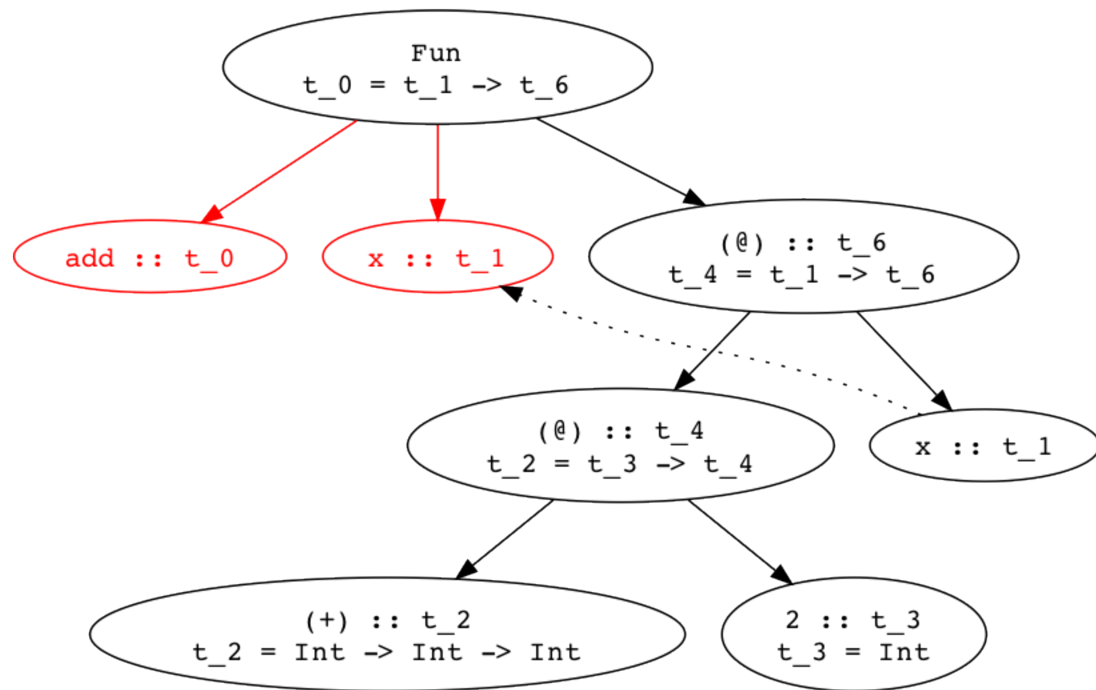

Figure 8.3: Parse tree labeled with type variables

Solving the generated constraints set using unification. The previous steps of the type inference algorithm have generated the following constraints set, written on the various nodes of the parse tree:

```
1)  t_0 = t_1 -> t_6
2)  t_4 = t_1 -> t_6
3)  t_2 = t_3 -> t_4
4)  t_2 = Int -> (Int -> Int)
5)  t_3 = Int
```

We now wish to find an assignment of actual types to type-variables in a way that will satisfy all above constraints. If such an assignment exists, we can say that the expression is well typed and that the type inference algorithm will end successfully.
To this end we will use 'Unification' which is a standard algorithm for solving systems of equations by a series of substitutions.
Here's how the unification algorithm will process our constraints set:
From Equations (3) and (4) above we get:

```
t_3 -> t_4 = Int -> (Int -> Int)
```

This implies the following:

```
6) t_3 = Int
7) t_4 = Int -> Int
```

From Equations (2) and (7) we get:

```
t_1 -> t_6 = Int -> Int
```

This implies:

```
8) t_1 = Int
9) t_6 = Int
```

Using all of the above conclusions, we get the following assignment of types to type-variables:

```
t_0 = Int -> Int
t_1 = Int
t_2 = Int -> Int -> Int
t_3 = Int
t_4 = Int -> Int
t_6 = Int
```

Since all constraints have been satisfied, we say that the given expression is well typed. The inferred type of the 'add' function equals to the value of type-variable `t_0` which is `Int->Int`.

Example 6

A Polymorphic Function Definifition. The type inference algorithm described above also supports polymorphic functions that are functions involving type-variables as arguments. In this example we will review the algorithm's execution on the 'apply' function which is defined as follows:

```
apply (f, x) = f x
apply :: (t ->t1, t) -> t1
```

Building the parse tree. We start by building the parse tree for the 'apply' function similarly to the previous example. In this example, we introduce a new type of node which is the 'Pair' node - representing the argument pair given to the function. As can be seen in figure 4 below, the root node indicates that this is a function parse tree. The root's children are the actual function name, a pair node representing the function's arguments and a third child of type 'application' which holds the parse tree of the function body. Nodes representing the same type variables are connected using a dashed line to ensure that they are assigned a single type variable in the next step.

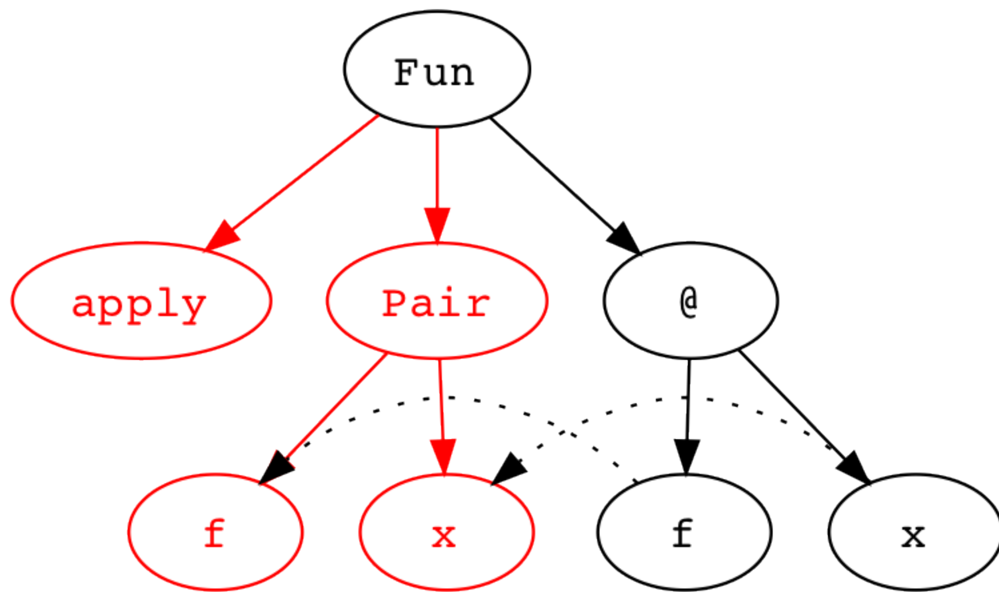

Figure 8.4: Parse tree for apply function

Assigning a type variable to the expression and each sub-expression. We now assign a type-variable to each node of the parse tree starting with the 'apply' node which is assigned with `t_0`, going through the tree leafs assigned with `t_1` and `t_2` traversing up till reaching the tree root. Note again that we will assign a single type-variable to nodes connected with a dashed line (nodes holding x and f will be assigned with type-variables `t_1` and `t_2` respectively).

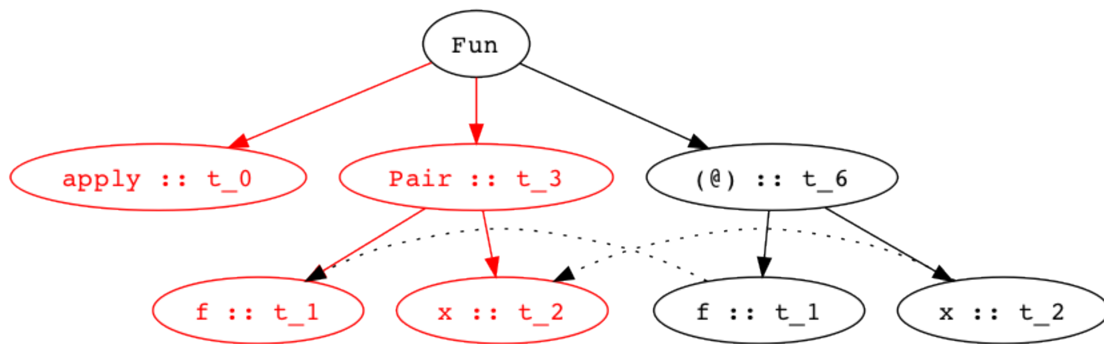

Figure 8.5: Parse tree for apply function labeled with type contraints

Generating a set of constraints. Based on the rules described in the previous example, for the application node ('@') we get the constraint `t_1 = t_2 -> t_6`, and for the

abstraction node ('Fun') we get the constraint `t_0 = t_3 -> t_6`. As for the new 'Pair' node, its type variable is composed of two type variables, giving in our case the constraint `t_3 = (t_1, t_2)`. All in all, we get the following constraint set:

```
1) t_1 = t_2 -> t_6
2) t_0 = t_3 -> t_6
3) t_3 = (t_1, t_2)
```

Solving the generated constraints set using unification. ¿From equations (2) and (3) above we get:

```
4) t_0 = (t_1, t_2)-> t_6
```

Using equation (1) we get:

```
5) t_0 = (t_2 -> t_6, t_2)-> t_6
```

Which is the type of the function, and can also be written in the same way the compiler wrote - `(t -> t1, t) -> t1`, which means that this function is polymorphic and may be applied to many different types.

Example 7

Application of a Polymorphic Function. In this example we calculate the type of the application of 'apply' to the ('add',3) argument pair, where 'add' is the function from the previous example, having a type of `'Int->Int'`. We repeat the steps of the algorithm as described earlier, this time using the known type of the given arguments.

Generating a set of constraints. Applying the rules we described in the previous examples, we generate the following set of constraints (where `a_1`, `a_2` denote a "fresh" type variable, different from all other type variables described in the constraint set):

```
1) t_1 = (a_1 -> a_2, a_1) -> a_2
2) t_2 = Int -> Int
3) t_3 = Int
4) t_4 = (t_2, t_3)
5) t_1 = t_4 -> t_5
```

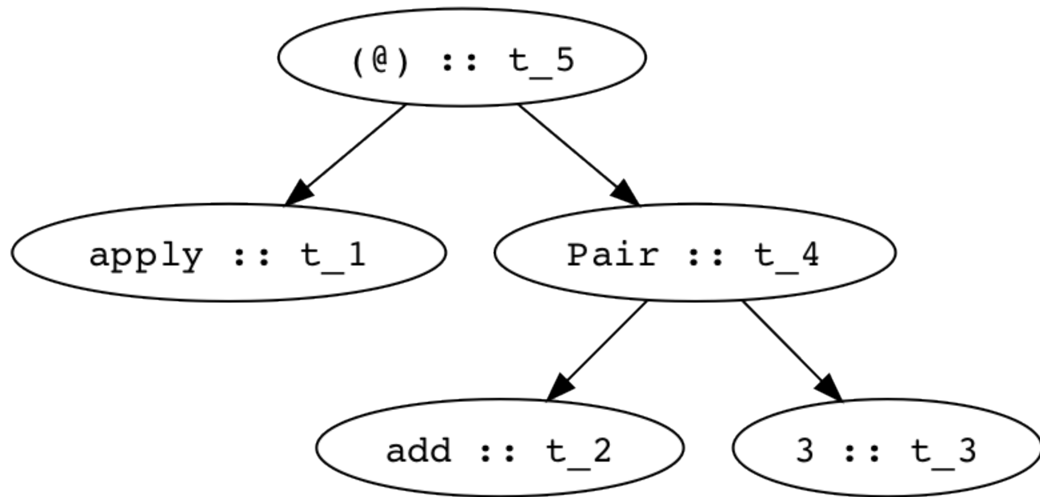

Figure 8.6: Parse tree for apply function labeled with type contraints

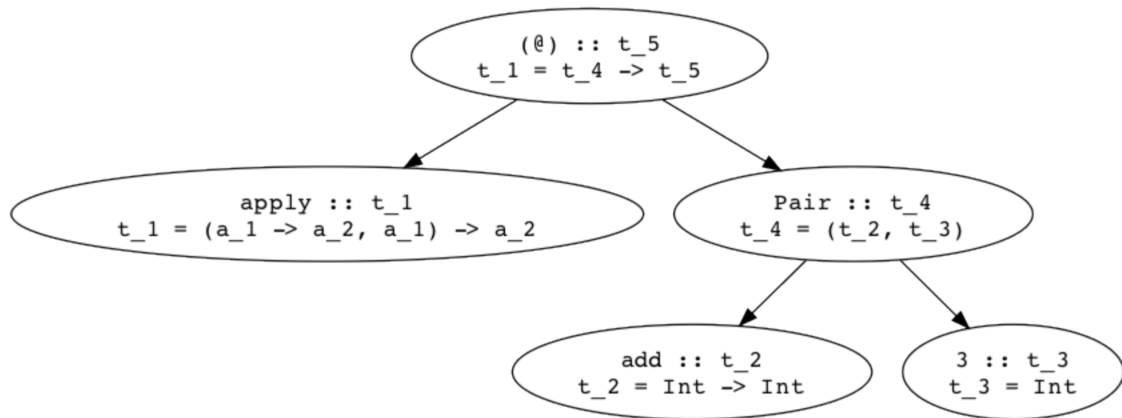

Figure 8.7: Parse tree for apply function labeled with type contraints

Solving the generated constraints set using unification. Using the unification algorithm to solve the equations set, we start with equations (1) and (5) and we get:

```
6) (a_1 -> a_2, a_1) -> a_2 = t_4 -> t_5
```

This can be separated into (corresponding parts of each expression should be equal):

```
7) (a_1 -> a_2, a_1) = t_4
8) a_2 = t_5
```

Next, from equations (4) and (7) we get:

```
9) a_1 -> a_2 = t_2
10) a_1 = t_3
```

And finally, by combining equations (2) and (9) we get:

```
11) a_1 = Int
12) a_2 = Int
```

To summarize, the following substitutions solve the equation set:

```
t_1 = (Int -> Int, Int) -> Int
t_2 = Int -> Int
t_3 = Int
t_4 = (Int -> Int, Int)
t_5 = Int
a_1 = Int
a_2 = Int
```

So a solution for the constraint set exists and thus the expression `apply(add,3)` has the type Int and it is well typed.

Example 8

A Function with Multiple Clauses. The type inference algorithm can also be applied to functions with multiple clauses such as the following 'concat' function:

```
concat ([], r) = r
concat (x:xs, r) = x : concat(xs, r)
concat :: ([t], [t]) -> [t]
```

In this case, the function 'concat' can be applied to any pair of lists, as long as both lists contain the same type of list elements.

To solve the type inference problem on this function with multiple clause s we run the algorithm for each clause separately, inferring its type:

```
concat :: ([t], t_1) -> t_1
concat :: ([t], t_1) -> [t]
```

Then, we impose the following constraint in order to satisfy the requirement that the two clauses must have the same type:

```
([t], t_1) -> t_1 = ([t], t_1) -> [t]
```

Which implies (second part of each equation side): `t_1 = [t]`.
And so the final type for the 'concat' function would be: `concat :: ([t], [t]) -> [t]`.

References

1. "Type Systems, Type Inference, and Polymorphism" (Chapter 6), Concepts in Programming Languages, John C. Mitchell, Cambridge Univ Press, 2003
2. `http://en.wikipedia.org/wiki/Type_inference`
3. `http://en.wikipedia.org/wiki/Unification_(computer_science)`
4. Unification in Prolog - `http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/3_1.html`