Semantic Analysis and Type Checking

Read Chapter 6 of the Dragon Book (you can skip sections 6.5, 6.6, and 6.7) or Chapter 6 of the Mogensen book.

Introduction

Semantic analysis takes as input the parse tree generated by the syntax analysis of the compiler, and checks that the program represented by that parse tree is meaningful. There are many checks that can be done by the semantic analysis, but the most common ones are: checking that every variable and function is defined before it is used, checking that every variable and function is not defined more than once, checking that every variable is initialized before it is used, and checking the types of expressions and statements.

In this course we are only going to look at type checking, since it is the most interesting and most important part of semantic analysis. In fact we will look only at a simple version of type checking, that uses synthesized attributes only and therefore can be computed by doing a simple bottom-up traversal of a parse tree. Since we want to keep the type checking simple, we will restrict ourselves to a simple programming language that looks somewhat like C or Pascal, and the most complicated programming language feature we will type check are functions. We will not look at more advanced features like Java classes or polymorphism (Java Generics) since type checking them cannot be easily done using attributes and requires instead to use specialized algorithms (type checking classes requires taking inheritance into account, type checking polymorphic methods requires being able to give multiple types to a single method depending on where and how it is used, etc.) In fact, to properly type check these more advanced features and understand the corresponding algorithms, we would have to study the formal theory of type systems, which would require us to first study the formal semantics of programming languages. See the book "Types and Programming Languages" by Benjamin Pierce if you are interested. In this course we will be happy to stick with simple features.

The goal of type checking is to compute a type for every expression and statement in a program (represented as a parse tree), and check that all these types are coherent with each other (for example, that the operands of the addition operator are integers or floating point numbers, but not function names). If some types are inconsistent then the type checker should inform the user about that. If all type are consistent then the parse tree is given to the next phase in

the compiler, which is the intermediate code generation. In some cases the type checker will also have to modify the parse tree to introduce type conversions (see below).

Before we look at type checkers themselves, we first need to look at types, type expressions, and type systems.

Types

A *type* is a syntactic description of a set of possible runtime values. "Syntactic" means that a type is simply syntax that a programmer writes in his source program. At compile time, the type checker inside the compiler has to give a meaning to that syntax. That meaning is a set of values: the values that the corresponding expression is allowed to have when the program is actually running. For example, consider this piece of code: `int x;`
The three characters "`int`" are just syntax that the programmer writes in his source program. When the type checker inside the compiler sees the corresponding token in the parse tree coming out of the syntax analysis, it determines that the variable `x` is only allowed to contain integers. The type checker will then verify that the program only stores integers in the variable `x`. This ensure that later, when the target program generated by the compiler is executed, the memory allocated for the variable `x` only contains bits representing integers and nothing else.

Note that types do not exist at runtime. At runtime the only thing that the microprocessors manipulates are bits in memory. These bits do not have a type. For example, the bits `0110111101` do not have a type, they are just bits. They can be bits for an integer, or a floating point number, or a string, or a pointer, or they can even be bits that represent microprocessor instructions. The microprocessor itself does not know what these bits are supposed to represent. The person who wrote the source program, and the compiler's type checker that analyzed that source program, are the only ones who know what kind of value those bits are supposed to represent. So if at runtime the microprocessor puts the bits `0110111101` into the memory allocated for the variable `x`, the microprocessor has no way to check whether this is the right thing to do or not. We know it is the right thing to do because the type checker has already verified at compile time that at runtime only the bits of integers will ever be stored in the memory allocated for `x`.

Some programming languages, like Perl, Python, or Scheme, do not have types, so a programmer never has to write something like "`int`" in a source program. Compilers for these programming languages therefore do not do any type checking at all. How does the language's implementation then verify that the program does not try to add an integer to a string, for example? This is done not using types but using tags: at runtime every value (i.e. every group of bits) has an extra tag (another group of bits) that describes what kind of value the program is dealing with. Before doing an addition, the addition operator will then check that the tags of its operands are the tags for integers and not

the tag for strings. Note that these tags essentially simulate types at runtime, but they are not types in the sense we have described above.

Basic Types and Constructed Types

There are two kinds of types.

- Basic types (or primitive types, or atomic types). These are the simplest types that cannot be split into smaller types. They typically correspond to the kinds of values that are directly supported by the microprocessor: `char`, `int`, `long`, `float`, `double`, `bool`, etc.
- Constructed types. These are the types that are made of various combinations of smaller types. Therefore they can be split into smaller types. For example: arrays, structures (records), pointers, etc.

Type Expressions

Types are just syntax in the source program. Internally the compiler needs to have its own representation of types that the type checker will then be able to manipulate, hence the need for type expressions. A *type expression* is the internal representation of a type inside the compiler. One of the jobs of the type checker is then to compute a type expression for each expression and each statement in the program being analyzed. To compute these type expressions, the type checker will start with the types specified by the user in his source program, transform them into type expressions, and then combine these type expressions in various ways using rules that we will describe below to get type expressions for all the various parts of the source program.

Because they initially are derived from types, type expressions essentially follow types, and are as follows.

- Basic type expressions: character, integer, floatingpoint, boolean, etc., that directly represent the corresponding basic types and which, like them, are atomic (cannot be split into subparts).
- The special basic type type_error, representing an error detected during type checking.
- The special basic type void, representing the absence of a type, which we will use for pieces of programs that do not compute a value (i.e. mostly statements).
- Type names, to be used with `typedef`s in C or with classes in Java. For example, the C code "`typedef foo int;`" creates a new name "`foo`" that a programmer can then use as type syntax in his program and which is a synonym for the integer type. Similarly the Java code "`class A {...}`" creates a new name "`A`" that a programmer can then use as a type when defining variables or method arguments.

- The constructed type expression array(i, t), which describes the type of an array with elements of type t and a set of index i. For example, the type expression corresponding to "`int[100]`" is "array(0...99, integer)". The set i of indexes is often a set of integers, but in some programming languages like Perl it can be a set of strings, etc.
- The constructed type expression pointer(t), which describes the type of a pointer to a value of type t. For example the type expression corresponding to "`int *`" is "pointer(integer)".
- The constructed type expression $T_1 \times T_2$ (the cartesian product of two type expressions T_1 and T_2) represents a pair of type expressions. This is useful for records and functions (see below).
- The constructed type expression record(name × type expression, ...) describes the type of a record (structure), where each member of the record is described by a pair that lists the name of the member and the associated type expression. For example, the type of the C structure "`struct{int i; char c;}`" is represented by the type expression "record(`i` × integer, `c` × character)".
- The constructed type expression $T_1 \rightarrow T_2$ describes the type of a function that takes a value of type T_1 as argument (the *domain* of the function) and returns a value of type T_2 as result (the *range* of the function). For example, the type if the C function "`int foo(char c){...}`" is represented by the type expression "character → integer". If the function takes several arguments, then they are grouped together using pairs. For example, the type of the C function "`int bar(char c, float f){...}`" is "(character × floatingpoint) → integer". As another example, the type expression "(integer → boolean) → (character → floatingpoint)" represents the type of a function that itself takes a function as argument and returns another function as result. The function taken as argument itself takes an integer as argument and returns a boolean as result. The function returned as result itself takes a character as argument and returns a floating point number as result. As a last example, the type of the addition operator for integers is represented by the type expression "(integer × integer) → integer", which means that the addition operator takes two integers as operands and returns an integer as result.

Type Systems

A *type system* is a set of mathematical rules (called *typing rules*) that describe how to compute type expressions for all the parts of a program (expressions, statements, definitions, etc.) Some rules will start with the types specified by the programmer in his source program and transform these types into equivalent type expressions, then other rules will take existing type expressions, check them, and combine them to get new type expressions for all the expressions and statements in the source program. All these rules must be *sound*: the rules

must guarantee that, if the source program contains a type error, then this type error will always be detected by the compiler. If the compiler guarantees to the user that all type errors will always be detected at compile time, then the programming language is said to be *strongly typed* or *type safe*.

So let's look at our first typing rules. Here is a small programming language that includes simple type definitions and expressions:

$$
\begin{array}{lll}
P \to D \texttt{ ; } E & T \to \texttt{char} & E \to \texttt{char\_lit} \\
D \to D \texttt{ ; } D & T \to \texttt{int} & E \to \texttt{num} \\
D \to T \texttt{ id} & T \to T \texttt{ *} & E \to \texttt{id} \\
 & T \to T \texttt{ [num]} & E \to E \texttt{ mod } E \\
 & & E \to \texttt{*} E \\
 & & E \to E \texttt{ [} E \texttt{]}
\end{array}
$$

In this language, a program is a sequence of variable definitions followed by a single expression. A variable definition is a type followed by the name of the variable. A type is either a basic type, like `char` or `int`, or a constructed type, like the pointer and array types. An expression is either a constant, like a character literal (e.g. `'a'` in C) or a number, or a variable name, or the modulo operation, or a pointer dereference, or an array element access. Note that this grammar is ambiguous (because of the $D \to D$ `;` D grammar production, and because no precedence or associativity is specified for the `mod`, `*`, and `[ ]` operators) but we do not care about this here: we are only interested in writing typing rules.

For most (but not all) of these different parts of a program we want to create typing rules that compute a synthesized "type" attribute which is going to be a type expression describing the type associated with that part of program.

Typing Rules for Types, Definitions, and Programs

Typing rules for basic types are easy:

$$
\begin{array}{ll}
T \to \texttt{char} & T.\text{type} := \texttt{char}.\text{type} \\
T \to \texttt{int} & T.\text{type} := \texttt{int}.\text{type}
\end{array}
$$

The question then is: what are the values of `char`.type and `int`.type? Since `char` and `int` are tokens and since the type attribute is synthesized, we know that `char`.type and `int`.type are intrinsic, so we have to find their values somewhere outside the semantic rules we are writing. In the present case these values do not come from the lexeme or the symbol table, as is often the case with intrinsic attributes. Rather the values of these two intrinsic attributes are defined in the reference manual for the programming language. For example, for the C language, the reference manual for C (i.e. the "ISO/IEC 9899:2011 - Programming Languages - C" standard) indicates that the three letters "`int`" appearing in a source program mean the integer type. That knowledge is then hard-coded directly into the semantic phase of the compiler so that `int`.type is a constant which always has the value "integer". Similarly for `char`.type which is defined once for all to be the type expression "character".

Here are the typing rules for constructed types, which are quite straightforward too:

$T_1 \rightarrow T_2$ `*`	T_1.type := pointer(T_2.type)
$T_1 \rightarrow T_2$ `[ num ]`	T_1.type := array(0... (`num`.val - 1), T_2.type)

The only tricky part in these two rules is to remember to do the "- 1" when computing the set of indexes for an array type, since we assume that in our little programming language the array indexes start at 0. Here obviously the synthesized attribute `num`.val is intrinsic and its value should be determined by looking at the corresponding lexeme. These rules then mean that, for example, "`int *`" has type pointer(integer) and that "`int[100]`" has type array(0...99, integer).

If you remember the addtype procedure from the previous lecture notes, then the following typing rule for variable definitions should be easy to understand:

$$D \rightarrow T \texttt{ id} \qquad \text{addtype}(\texttt{id}.\text{entry}, T.\text{type})$$

Such a semantic rule simply takes the type expression computed for T and uses the addtype procedure to store that type expression into the symbol table entry for the identifier. Note that the definition D itself does not have a type, the corresponding semantic rule simply assigns a type to a variable. It makes sense for variable definitions to not have a type since they do not compute a value.

The following two grammar productions do not have any semantic rule associated with them:

$$P \rightarrow D \texttt{ ; } E$$
$$D \rightarrow D \texttt{ ; } D$$

That's because, as we just noted above, definitions do not have types, and in fact neither do whole programs.

Typing Rules for Expressions

We can now look at typing rules for expressions. Just like basic types, constant expressions are easy:

$E \rightarrow$ `char_lit`	E.type := `char_lit`.type
$E \rightarrow$ `num`	E.type := `num`.type

Again, just like for the basic types `char` and `int`, the values of `char_lit`.type and `num`.type are intrinsic and defined once for all by the reference manual of our little programming language to be character and integer, respectively.

When an expression is simply a variable name, we use a lookup function that looks at the symbol table entry for the variable and returns the corresponding type, which was store there beforehand using the addtype procedure when the variable was defined (this in turn explains why variables always have to be defined before they are used, otherwise type checking variables would not be possible in a single parse tree traversal):

$$E \rightarrow \texttt{id} \qquad E.\text{type} := \text{lookup}(\texttt{id}.\text{entry})$$

The typing rule for the modulo operator is much more interesting. We know that this operator, which computes the integer remainder of an integer division, always returns an integer as a result. But before we compute the type of the result, we must first check the types of the operands since the modulo operation only makes sense if both operands are integers. So if both operands are integers then the result of the operator is in integer, but if either (or both) operands are not integers then we have detected a type error, which we indicated using the special type type_error:

$E_1 \rightarrow E_2$ `mod` E_3 if(E_2.type == integer and E_3.type == integer)
then E_1.type := integer
else E_1.type := type_error

For example, the type of the expression "`2 mod 3`" is integer, since both operands have type integer, while the type of the expression "`2 mod 'a'`" is type_error, since the second operand is a character literal.

Similarly for pointer dereference expressions: before dereferencing a pointer we must first make sure that we have indeed a pointer to dereference. Trying to dereference, say, an integer, must be a type error. If indeed we have a pointer, then the type of the dereference expression is the type of the value the pointer points at:

$E_1 \rightarrow$ `*` E_2 if(E_2.type == pointer(t))
then E_1.type := t
else E_1.type := type_error

For example, if a variable `x` has type pointer(integer) then the expression "`*x`" has type integer. If a variable `y` has type character, then the expression "`*y`" has type type_error.

Note that this typing rule never tries to look at the value of the expression E_2, it only ever looks at its type. This explains why such a typing rule cannot detect `NULL` dereferences. For example, in C, if a variable `x` has type pointer(integer) and value `NULL`, then the C type system will just compute that the expression "`*x`" has the type integer and the program will compile without error. It is only when running the program that the bug will be detected (hopefully, since C is not safe...) This is also true in Java, even though Java does have a strong type system. The Java type system will not detect the error and an exception will happen at runtime (which is at least guaranteed to happen, since Java is safe). So type systems do not detect all errors, they only detect errors related to types, not errors related to values (like the `NULL` value). Similarly, our typing rule for the modulo operator above does not detect the erroneous case when the second operand has the value zero, for which the modulo operation is undefined.

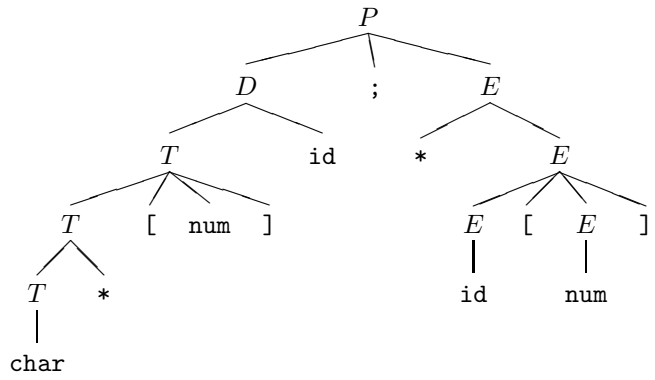
Finally, we have array accesses. Here there are two checks to make before computing the type of the expression. First we need to check that we have indeed an array. Second we have to check that the expression that tries to access an element of an array uses only integer indexes. If either of these two checks fails then we have a type error. Otherwise the array access succeeds and the type of the resulting expression is the type of a single element of the array:

$E_1 \rightarrow E_2$ `[` E_3 `]`	if(E_2.type == array(i, t) and E_3.type == integer) then E_1.type := t else E_1.type := type_error

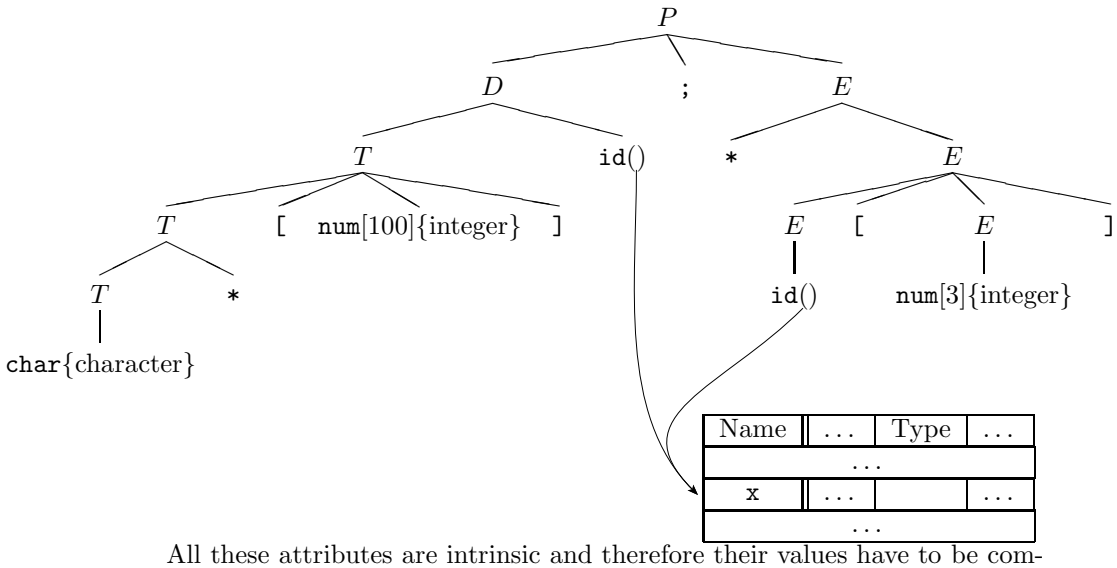
For example, if a variable `x` has type array(0...99, character) then the expression "`x[3]`" has the type character, since `x` has an array type and `3` is an integer. The type of the expression "`2[3]`" is type_error, since `2` is not an array, and the type of the expression "`x['a']`" is type_error too since the index used in the expression is not an integer.

Note that in the typing rule above, the set of indexes i plays no role. This means that the type checking does not attempt to check that array accesses are within the bounds of the array. For example, if a variable `x` has type array(0...99, character) then the expression "`x[333]`" has the type character and no error will be detected by the compiler. Again, type checkers only look at types, not at values, so they do not attempt to check whether the value `333` of the index is in the set 0...99 of allowed array indexes.

Here is a complete example showing how the various typing rules combine various type expressions to give the type of a complete expression. Suppose we have the following source code: `char *[100] x; *x[3];`
Here is the corresponding parse tree (assuming that array access has higher precedence than pointer dereference, which is the case in C):



Since all the attributes in our semantic rules are synthesized, we can use any bottom-up order to compute them (as long as the definition of `x` is analyzed before the use of `x` to ensure the symbol table contains the type of `x` when we need it — in practice we would use the same bottom-up left-right order that is also used by the bottom-up parser that constructs the parse tree, so that parsing and type checking can happen at the same time). So we can for example start with computing the val attribute of the `num` tokens (shown here between brackets), the entry attributes of the `id` tokens (shown here between parentheses), and the type attribute of the `char` token and `num` tokens (shown here between braces):

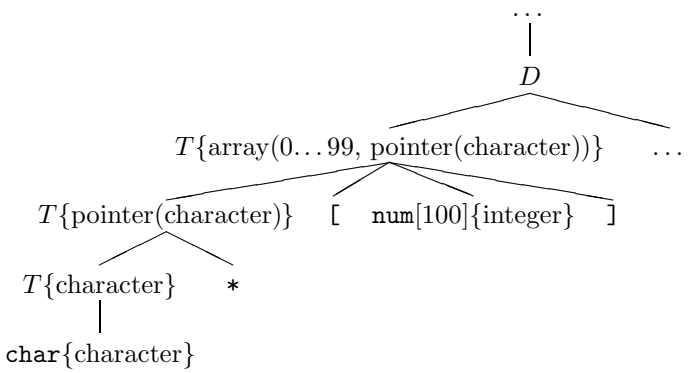


All these attributes are intrinsic and therefore their values have to be computed without looking at the typing rules. In the present case, the values of the attributes `num`.val come from looking at the corresponding lexemes, the values of the `id`.entry attributes come from searching the symbol table for the entry corresponding to the lexeme `x` associated with each `id` token, and the values of the `char`.type and `num`.type attributes are hard-coded in the compiler to always be character and integer respectively.

Next we use the three typing rules:

$T \rightarrow$ `char`	T.type := `char`.type
$T_1 \rightarrow T_2$ `*`	T_1.type := pointer(T_2.type)
$T_1 \rightarrow T_2$ `[ num ]`	T_1.type := array(0...(`num`.val - 1), T_2.type)

in that order to compute the type attributes of the three T nonterminals in the parse tree:



Note that we never use the attribute `num`.type in the computation above, so computing the "integer" value of that attribute in the parse tree just above

was actually a bit useless, but it does not hurt us... (in fact, we ought to use `num`.type to check that `num` really is an integer before computing `num`.val - 1 but our simple typing rule for array definitions does not do that).

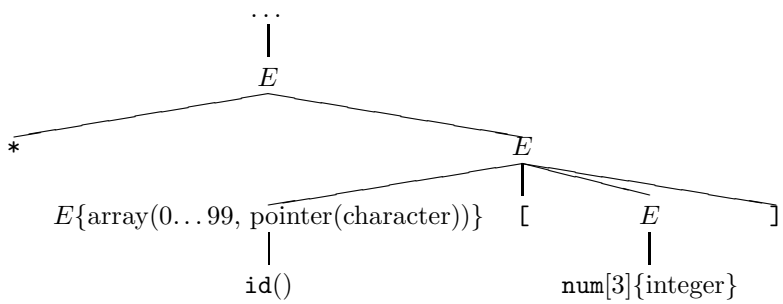
In the next step up in the computation, at the level of the D, we use the value array(0...99, pointer(character)) of the T.type attribute and the value (a pointer into the symbol table) of the `id`.entry attribute to update the symbol entry for `x` with the type expression we just computed:

Name	...	Type	...
...			
`x`	...	array(0...99, pointer(character))	...
...			

Once the symbol table has been updated, we can do the bottom-up computation of the attributes in the right side of the parse tree. First we have to use the following typing rule:

$$E \rightarrow \texttt{id} \qquad E.\text{type} := \text{lookup}(\texttt{id}.\text{entry})$$

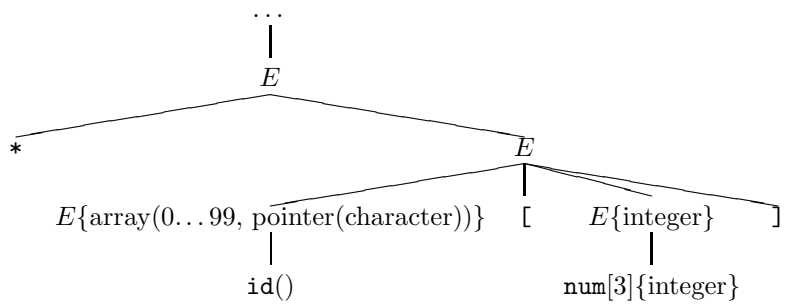
and the lookup function to check the type of the identifier. Since `id`.entry points to the same symbol table entry into which we saved the type expression for `x` at the previous step, the lookup function is going to return that same type expression as a result:



We then use the semantic rule:

$$E \rightarrow \texttt{num} \qquad E.\text{type} := \texttt{num}.\text{type}$$

to compute the type expression for the index expression of the array access expression:

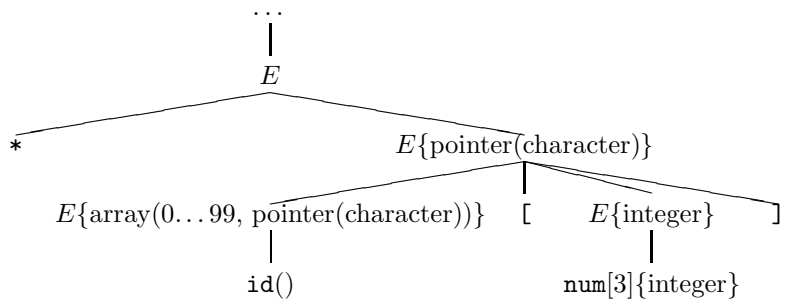


Note that here we never use the attribute num.val (i.e. the value 3). This is what we expect since, as we explained before, our typing rules do not do any array bounds checking.

Next we use the typing rule for array accesses:

$E_1 \rightarrow E_2$ [E_3] if(E_2.type == array(i, t) and E_3.type == integer)
then E_1.type := t
else E_1.type := type_error

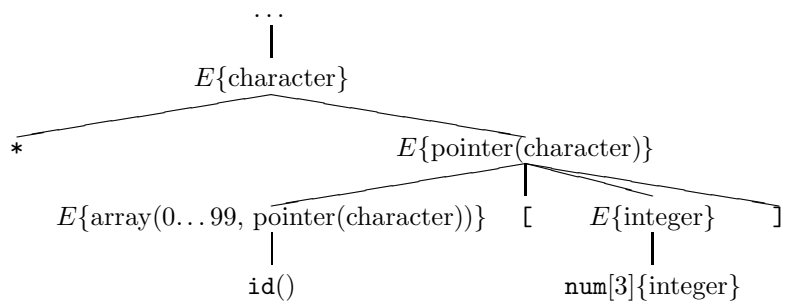
Since E_2.type is array(0...99, pointer(character)) then the test "E_2.type == array(i, t)" is true, with the set of indexes i being 0...99 and the type t of the elements being pointer(character). Since E_3.type is integer then the test "E_3.type == integer" is true too. Therefore the type of the whole array access expression is t, which, as we just indicated, is pointer(character):



Finally we use the typing rule for pointer dereferences:

$E_1 \rightarrow$ * E_2 if(E_2.type == pointer(t))
then E_1.type := t
else E_1.type := type_error

Since we just computed that E_2.type is pointer(character), the test "E_2.type == pointer(t)" is true, with the type t being character. Therefore the type of the whole pointer dereference expression is t, which is character:



The last step is to compute the attributes for the root P of the parse tree, but there are no such attributes, so we are done, and we then know that the expression in the program computes a character.

There are two additional interesting things to note here.

- If you look at how we have computed the different type expressions in the parse tree, we started with the type "`char *[100]`" that the user wrote in his source program, transformed that type into a type expression using the typing rules for basic and constructed types, then transformed that type expression in various ways using the typing rules for expressions to get type expressions for all expressions and sub-expressions in the program. This is typical of type checking: the types specified by the user in his source program serve as starting point for the type checking, which then computes type expressions for the other parts of the program, checking for type errors in the process.

- Note how using the symbol table to store the type expression for `x` allowed us to transfer information from the top of the left sub-tree (the sub-tree concerned with user-written types) to the bottom of the right sub-tree (the sub-tree concerned with expressions). If we had not used the symbol table, then the only way to transfer information from the left sub-tree to the right sub-tree would have been to use an extra *inherited* attribute and have extra typing rules to propagate the type expression for `x` down the right sub-tree. Obviously this would have made the whole computation of attributes much more complicated. It would also have prevented us from using only a strictly bottom-up order for the computation (in other words, in practice it would prevent us from doing the type checking directly at the same time as a bottom-up parser is building the parse tree). So using the symbol table simplifies our typing rules a lot. This is fortunate, because we need to store the type expressions for variables anyway because the subsequent phases of the compiler will need that information (the code generation, in particular, to do proper instruction selection).

Booleans

What if we now want to extend our little programming language to include booleans and boolean operations? This is easy to do. First we just add a new

basic type `bool`:

$$T \rightarrow \texttt{bool} \qquad T.\text{type} := \texttt{bool}.\text{type}$$

with the intrinsic attribute `bool`.type being set once for all in the compiler to be the type expression boolean.

Next we can add new expressions for boolean literals (constants). Since there are only two possible boolean literals, we can have specific tokens for them:

$$\begin{array}{ll} E \rightarrow \texttt{true} & T.\text{type} := \texttt{true}.\text{type} \\ E \rightarrow \texttt{false} & T.\text{type} := \texttt{false}.\text{type} \end{array}$$

with the intrinsic attributes `true`.type and `false`.type being always the type expression boolean.

Now we just have to add new expressions for operators that manipulate booleans. There are two kinds of such operators. First we have relational operators `relat_op` (i.e. `<=`, `<`, `==`, `>`, `>=`, and `!=`) that compare integers and return a boolean:

$$\begin{array}{ll} E_1 \rightarrow E_2 \ \texttt{relat\_op} \ E_3 & \text{if}(E_2.\text{type} == \text{integer and } E_3.\text{type} == \text{integer}) \\ & \text{then } E_1.\text{type} := \text{boolean} \\ & \text{else } E_1.\text{type} := \text{type\_error} \end{array}$$

This typing rule is very similar to the one for `mod`, except that the result is a boolean type expression, not an integer one.

We also have boolean operators `bool_op` (i.e. `and`, `or`, `xor`) that combine two booleans into a single one:

$$\begin{array}{ll} E_1 \rightarrow E_2 \ \texttt{bool\_op} \ E_3 & \text{if}(E_2.\text{type} == \text{boolean and } E_3.\text{type} == \text{boolean}) \\ & \text{then } E_1.\text{type} := \text{boolean} \\ & \text{else } E_1.\text{type} := \text{type\_error} \end{array}$$

This typing rule is very similar to the previous one, except that the operands now have to be booleans.

Finally we have the unary `not` operator, which only takes one operand:

$$\begin{array}{ll} E_1 \rightarrow \texttt{not} \ E_2 & \text{if}(E_2.\text{type} == \text{boolean}) \\ & \text{then } E_1.\text{type} := \text{boolean} \\ & \text{else } E_1.\text{type} := \text{type\_error} \end{array}$$

Functions

We now want to add functions to our programming language. So we need to add two things: syntax for function definitions, and syntax for function application.

For function definitions, we will use the following grammar production:

$$D \rightarrow T \ \texttt{id} \ \texttt{(} \ T \ \texttt{id} \ \texttt{)} \ E$$

Here the first (leftmost) identifier is the name of the function, and the type on its left is the type returned by the function. The type and identifier inside the parentheses are for the formal argument of the function. To simplify the problem, we limit ourselves to function that always take exactly one argument. Again, to simplify, we decided that the code of the function (the function's body) is going to be a single expression E. For example, the grammar production above matches the following function definition:

```
bool foo(int i) i <= 3
```

Here `foo` is the name of the function, the return type of the function is `bool`, the name of the formal argument is `i`, the type of the formal argument is `int`, and the code (body) of the function is the single expression "`i <= 3`".

For function application, we are going to use the following grammar production:

$$E \to E \texttt{ (} E \texttt{)}$$

Here the first expression should evaluate to a function, and the second expression inside the parentheses is going to compute the value of the actual argument that is given to the formal argument of the function being applied. For example, the grammar production above matches the following function application:

```
foo(2)
```

where `foo` is an expression which is the name of the function we want to apply, and `2` is the actual argument given to the function.

So let's start writing typing rules for these two new grammar productions. Function definitions are interesting to type check because they introduce the notion of *scope*. For example, given the code:

```
bool foo(int i){ ... i ... }
```

the integer `i` can only be used inside the code of the function, not outside. Since we are using the symbol table to store the type expressions of all variables, this means we will have to add the type expression for `i` to the symbol table when we start analyzing the code of the function `foo` but then we will have to remove the type expression for `i` from the symbol table once we are done analyzing the code of `foo`. If we leave the type expression for `i` in the symbol table even after we are done analyzing the code of the function `foo` then suddenly the variable `i` which is outside the function `foo` in this code:

```
bool foo(int i){ ... i ... } ... i ...
```

will have a type and the compiler will then not detect the error.

So the first thing our typing rule for a function definition should do is take care of the scope of the formal argument of the function:

$$D \to T_1 \texttt{ id}_1 \texttt{ (} T_2 \texttt{ id}_2 \texttt{)} \; E \qquad \begin{cases} \text{addtype}(\texttt{id}_2\text{.entry}, T_2\text{.type}) \\ \dots \\ \text{removetype}(\texttt{id}_2\text{.entry}) \end{cases}$$

where the addtype procedure adds the type expression T_2.type to the symbol table entry for $\texttt{id}_2$ to start the scope of the formal argument, and removetype then removes the same type expression from the same symbol table entry to end the scope of the formal argument.

Once the formal argument is in scope, we can then compute the type attribute of the body E of the function. The type expression we compute for E should then match the type T_1 that the function returns. In other words, if the definition of the function says that the function returns a value of type T_1 then the body E of the function should actually compute a value of that exact same type. So we need to check this in our typing rule:

$$D \to T_1\ \texttt{id}_1\ \texttt{(}\ T_2\ \texttt{id}_2\ \texttt{)}\ E \qquad \begin{cases} \text{addtype}(\texttt{id}_2\text{.entry}, T_2\text{.type}) \\ \text{if}(E\text{.type} == T_1\text{.type}) \\ \text{then} \ldots \\ \text{else} \ldots \\ \text{removetype}(\texttt{id}_2\text{.entry}) \end{cases}$$

If the type expression of the body E of the function matches the type expression that the function should return, then we need to compute a type expression for the function itself. Remember that a type expression for a function is of the form $X \to Y$ where X is the type expression of the formal argument of the function and Y is the type expression of the value returned by the function. In the present case, we know what the type expression of the formal argument is: T_2.type. We also know what the type expression of the value returned by the function is: T_1.type. Therefore the type of the whole function is: T_2.type $\to$ T_1.type. If the type expression of the body E of the function does not match the type expression that the function should return, then we have a type error:

$$D \to T_1\ \texttt{id}_1\ \texttt{(}\ T_2\ \texttt{id}_2\ \texttt{)}\ E \qquad \begin{cases} \text{addtype}(\texttt{id}_2\text{.entry}, T_2\text{.type}) \\ \text{if}(E\text{.type} == T_1\text{.type}) \\ \text{then} \ldots T_2\text{.type} \to T_1\text{.type} \ldots \\ \text{else} \ldots \text{type\_error} \ldots \\ \text{removetype}(\texttt{id}_2\text{.entry}) \end{cases}$$

The question then is, what do we do with these two types expressions T_2.type $\to$ T_1.type and type_error? The answer is simple: since we are defining a function, since that function has a name represented by the token $\texttt{id}_1$, and since every name should have an entry in the symbol table that specify its associated type expression, we simply use the addtype procedure to store the type expressions we just computed in the symbol table entry for the function's name:

$$D \to T_1\ \texttt{id}_1\ \texttt{(}\ T_2\ \texttt{id}_2\ \texttt{)}\ E \qquad \begin{cases} \text{addtype}(\texttt{id}_2\text{.entry}, T_2\text{.type}) \\ \text{if}(E\text{.type} == T_1\text{.type}) \\ \text{then addtype}(\texttt{id}_1\text{.entry}, T_2\text{.type} \to T_1\text{.type}) \\ \text{else addtype}(\texttt{id}_1\text{.entry}, \text{type\_error}) \\ \text{removetype}(\texttt{id}_2\text{.entry}) \end{cases}$$

Let's look at our example function definition again:

```
bool foo(int i) i <= 3
```

Here T_1 is `bool`, the token $\texttt{id}_1$ represents the name `foo` of the function, T_2 is `int`, the token $\texttt{id}_2$ represents the name `i` of the formal argument of the function, and E is the expression "`i <= 3`".

We then apply the typing rule. Since T_2 is `int`, therefore T_2.type is integer. Since the token $\texttt{id}_2$ represents the name `i` of the formal argument of the function, therefore $\texttt{id}_2$.entry points to the symbol table entry for the name `i` (but only while `i` is in scope). We then use the addtype procedure to store the type expression integer in the symbol table entry for `i`:

Name	...	Type	...
...			
`i`	...	integer	...
...			

Once that is done, we have to type check the body of the function itself. Since E is the expression "`i <= 3`", we use the semantic rule for variable references to look up the type of `i` from the symbol table and we get the type expression integer. We then use the semantic rule for integer constants (represented by the token `num`) to get the type expression for `3`, which is integer too. We then use the semantic rule for the `<=` relational operator: since the type expressions for both operands are integer there is therefore no type error and the result is then a boolean. So E.type is boolean.

We now have to compare E.type and T_1.type. We know that E.type is boolean. Since T_1 is `bool` we know from the typing rule for `bool` that T_1.type is boolean. Therefore E.type (the type expression of the body of the function) is the same as T_1.type (the type expression for the result of the function) and there is no type error anywhere in the whole function.

Since T_2.type is integer and T_1.type is boolean we then compute that the type expression for the whole function is integer $\rightarrow$ boolean. We then use the addtype procedure to store this type expression in the symbol table entry for the name `foo` of the function:

Name	...	Type	...
...			
`i`	...	integer	...
...			
`foo`	...	integer $\rightarrow$ boolean	...
...			

Once we have finished type checking the function, we still need to use the removetype procedure to remove the type of `i` from the symbol table, since the variable `i` should not be visible outside the function:

Name	...	Type	...
...			
`foo`	...	integer → boolean	...
...			

And we are then done type checking the function definition.

Writing a typing rule for function application is much easier. The only thing we have to do is to check that we are indeed applying a function, and that the type of the actual argument matches the type of the formal argument of the function. If all that is true, then the type of the whole application is the type returned by the function as a result:

$E_1 \rightarrow E_2$ `(` E_3 `)` if(E_2.type == s → t and E_3.type == s)
then E_1.type := t
else E_1.type := type_error

What this typing rule means is this: if the expression E_2 evaluates to a function that takes a value of type s as argument and returns of value of type t as result, and if the expression E_3 for the actual argument evaluates to a value with the same type s that the function expects for its argument, then the result of applying the function to the actual argument is a value with type t (the type for the values returned by the function as result).

For example, using again the same function `foo` as above, if we type check the application:

```
foo(2)
```

then E_2 is the function name `foo` and the actual argument E_3 is `2`. Since E_2 is a name, we use the semantic rule for variable references to look up the type of `foo` from the symbol table and we get the type expression we computed before when we analyzed the definition of `foo`: integer → boolean. The typing rule for function application says that E_2 must be a function with a type expression of the form s → t, which is exactly the kind of type expression we have for `foo`, with the domain s being integer and the range t being boolean. We then use the semantic rule for integer constants (represented by the token `num`) to get the type expression for `2`, which is integer. We then compare the type expression integer for the formal argument E_3 with the type expression s of the domain of the function. Since s is integer too, the type expression of the actual argument matches the type expression of the formal argument of the function, and therefore there is no type error. The type expression for the whole application is then the type expression t of the range of the function. In other words the result of the application "`foo(2)`" is a boolean.

If, in the function application, the expression E_2 is not a function (for example: "`7(2)`") or if the type of the actual argument does not match the type of the formal argument of the function (for example: "`foo('a')`") then we have a type error.

One last point to note: if you look at the typing rules for function definitions and function applications that we wrote above, you will see that the typing rule

for function definitions *introduces* a function type expression (a type expression of the form s $\rightarrow$ t is created) while the typing rule for function applications *eliminates* a function type expression (a type expression of the form s $\rightarrow$ t disappears). This is a general rule for all type expressions (integers, characters, arrays, pointers, functions, etc.): there is always at least one typing rule to introduce the type expression and at least one typing rule to eliminate the same kind of type expression. For example, for pointers, there is one typing rule to introduce type expressions of the form pointer(t) (the typing rule associated with the grammar production "$T \rightarrow T$ `*`") and one typing rule to eliminate type expressions of the same form pointer(t) (the typing rule associated with the grammar production "$E \rightarrow$ `*` E"). Therefore if you create a new kind of type expression (for classes, for example) you know you will have to write at least two new typing rule for that new kind of type expression.

Typing Rules for Statements

The difference between expressions and statements is that expressions compute values and have no side effect, while statements do not compute values and have only side effects (modifying the content of the memory, printing something to the screen, sending a packet on the network, etc.) Since a type represents a set of possible values, and since statements do not compute any value, the typing rules for statements will use a special "void" type to indicate that the set of possible runtime values computed by a statement is always empty. The void type in essence is just a syntactic notation for the empty set of runtime values.

If we want to add statements to our little programming language then we first need to add a new grammar production for P:

$$P \rightarrow D \ ; \ S$$

so that a program can be a sequence of definitions followed by a statement (represented by the new nonterminal S).

We can then add to the language our first kind of statements, which are assignments:

$$S \rightarrow \texttt{id} = E$$

if(lookup(`id`.entry) == E.type)
then S.type := void
else S.type := type_error

Here we first get the type expression for the identifier from the corresponding entry in the symbol table, then we compare the type expression for the identifier with the type expression for the expression. This check is to prevent assigning a value of the wrong type to the variable represented by the identifier. For example, the code "`x = true`" should be a type error if the variable `x` was defined to be an integer. If the test is true, then the assignment is correct and the type of the whole statement is void (i.e. the statement computes the value of the expression and then modifies the value of the variable, but the statement itself does not return a value). Otherwise we have a type error.

Next we can add a conditional statement:

$$S_1 \rightarrow \texttt{if}\ E\ \texttt{then}\ S_2 \qquad \begin{array}{l}\text{if}(E.\text{type} == \text{boolean})\\ \text{then } S_1.\text{type} := S_2.\text{type}\\ \text{else } S_1.\text{type} := \text{type\_error}\end{array}$$

To type check a conditional statement, the first thing we do is check that the test expression computes a boolean. This is for example what happens in the Java or Pascal programming languages, where the test expression of an `if` statement always has to compute a boolean. In the C programming language the test expression can compute a value of any type, so the typing rule for conditional statements in the C language does not do this check at all, it simply ignores the value of E.type. If the value computed by E is not a boolean, then we have a type error. If the value computed by E is a boolean, then the type expression S_1.type for the whole statement depends on the type expression S_2.type for the `then` part of the statement. If S_2.type is void, then everything is fine and S_1.type can be void too. If S_2.type is type_error, then we need to propagate this error one level up and therefore S_1.type should be type_error too. In short, S_1.type should always be the same as S_2.type.

Next let's add an iteration statement to our language:

$$S_1 \rightarrow \texttt{while}\ E\ \texttt{do}\ S_2 \qquad \begin{array}{l}\text{if}(E.\text{type} == \text{boolean})\\ \text{then } S_1.\text{type} := S_2.\text{type}\\ \text{else } S_1.\text{type} := \text{type\_error}\end{array}$$

Here the typing rule is the same as the one for the conditional statement above: we first check that the test expression of the `while` loop computes a boolean, and of it does then the type expression for the whole loop statement is the same as the type expression for the `do` part of the loop (i.e. either void or type_error, depending on what S_2.type is).

The reason the typing rules for the `if` statement and for the `while` statement look so similar is because the type system does not care how many times the statement S_2 will be executed at runtime (zero or one time for the `if` statement, zero to an infinite number of time for the `while` statement), it only cares about the type of that statement S_2 (and about the type of E, of course).

To finish with statement, let's add some new syntax to have sequences of statements:

$$S_1 \rightarrow S_2\ \texttt{;}\ S_3 \qquad \begin{array}{l}\text{if}(S_2.\text{type} == \text{void and } S_3.\text{type} == \text{void})\\ \text{then } S_1.\text{type} := \text{void}\\ \text{else } S_1.\text{type} := \text{type\_error}\end{array}$$

Here we simply check that both S_2 and S_3 are void (i.e. do not contain any type error), in which case the whole sequence has type expression void too (i.e. does not contain any type error). If either S_2 or S_3 (or both) are a type error, then the whole sequence is a type error too.

Type Systems and Mathematical Logic

Let's look again quickly at the typing rule for function application:

$E_1 \rightarrow E_2$ (E_3) if(E_2.type == s $\rightarrow$ t and E_3.type == s)
then E_1.type := t
else E_1.type := type_error

Now let's write the same rule again, after erasing everything that is related to the grammar symbols used in the grammar production or that is related to errors. We get:

if(... s $\rightarrow$ t and ... s)
then ... t

Then let's rewrite the same thing in a slightly different way:

$$\frac{\text{s} \rightarrow \text{t} \quad \text{s}}{\text{t}}$$

What this means is that, if we have a function of type s $\rightarrow$ t and an argument of type s then applying the function to the argument gives us a value of type t.

Now let's look at the following rule from mathematical logic:

$$\frac{A \Rightarrow B \quad A}{B}$$

This logic rule is called *modus ponens* and it means this: if (above the line) A implies B and A is true then (below the line) B is true (where A and B are any proposition from mathematical logic).

If you compare the modus ponens rule with our simplified typing rule for function application, you can easily see that they have the exact same shape! In fact this is true for all the typing rules we have written: every typing rule we have seen has in fact a corresponding rule in mathematical logic! As another example, we said towards the beginning of these lecture notes that the attribute `int`.type is intrinsic and is hard-coded in the compiler to always be the type expression integer. This is equivalent to saying that the mathematical proposition "integer" is an axiom (something that is always true and does not need to be proven) in the logic theory corresponding to our type system.

Therefore the whole type system we have created above (all the typing rules for types, expressions, statements, etc, all together) is equivalent to some form of mathematical logic. This relationship between type systems and logic is called the Curry-Howard Isomorphism (or the Curry-Howard Correspondence) and is extremely important in computer science because it means that we can then start studying type systems as if they were pure mathematical objects. In turn that means that we can prove mathematical theorems about our type systems. In particular it becomes possible to prove mathematically that the type system above (all the typing rules we wrote) is correct! In fact this is what *strongly typed* (or *type safe*) actually means for a programming language, from a formal point of view: it means that the programming language (like Java) has a type system which is equivalent to some logic theory and which has then been mathematically proven to be correct. That is why we know that Java's type system is guaranteed to always find all type errors: because someone

actually proved mathematically that Java's type system is correct (or "sound" as logicians say). No one has been able to do this for the C programming language because the typing rules for C are too weak (e.g. any type can be changed into any other type simply by using an explicit cast in the source program) to correspond to any correct logic theory.

If you want more information about various type systems and the Curry-Howard Isomorphism, I strongly recommend the book "Types and Programming Languages" by Benjamin Pierce.

Defining Type Systems

Every programming language has a single type system (a set of typing rules) which is usually defined in the reference manual for that programming language. In most reference manuals (the one for Java, for example) the rules are written in plain English, because it makes it easier to write, read, and understand the rules compared to writing the rules using some sort of mathematical notation (like we have done above, using attributes). Some programming languages though (most notably ML) have typing rules that are only written using some mathematical notation (usually based on the notation from mathematical logic, because of the Curry-Howard Isomorphism, rather than based on attributes as we have done). Mathematical rules are harder to write and read, but they have the huge advantage over English of being unambiguous. When typing rules are written in English, it is much easier to write a rule that fails to specify exactly what is supposed to happen in some special cases, simply because human languages are not very precise. Different people reading the same English language for the same typing rules might then understand the rules in different ways, which then means that they will give different meanings to the same source programs. Obviously we want everyone to always agree on the meaning of programs, so you have to be careful about ambiguities in the text of reference manual for programming languages...

Type Checkers

Now that we know about type systems, it is easy to understand what a type checker is: a *type checker* is simply some piece of software inside the semantic analysis of a compiler that implements all the typing rules for the programming language that the compiler accepts as source language. In short, a type checker is just an implementation of a type system.

If the source program (represented as a parse tree) given to the type checker of a compiler follows all the rules of the type system implemented by the type checker then the program is deemed correct (from the point of view of types, at least) and the semantic analysis can then give the parse tree to the next phase of the compiler (the intermediate code generation). If the type checker discovers that the (parse tree for the) source program does not follow all the typing rules that the type checker implements, then the type checker outputs

an error message to the user, continues to type check the rest of the parse tree to try to find more type errors (usually compilers do not try to do much error recovery to try to guess what the correct type should have been after they have found a type error, they simply use the special type type_error, print an error message, and keep going) and then aborts without giving the parse tree to the next compiler phase.

As we have written above, a given programming language has a single type system. The typing rules for that system are written once for all in the reference manual for that programming language. A programming language will usually have many type checkers though: every implementation of the programming language (compiler, interpreter, as well as various static analyzers) will have its own implementation of a type checker in it and therefore its own implementation of the programming language's type system.

For example, there is only one type system for Java, which is defined (in English) in the reference manual for Java ("The Java Language Specification", by Gosling, Joy, Steele, Bracha, and Buckley). There are many type checkers for Java though: every compiler or interpreter for Java contains a type checker, and each of these type checkers implements the same typing rules described in the Java reference manual.

Of course, if you wanted to, you could create your own typing rules for Java and implement those rules in the type checker of your own compiler for Java. But if your new rules were not completely equivalent to the typing rules defined in the Java reference manual, then your programming language would not be Java anymore: it would look like Java (i.e. the syntax would be the same) but the meaning would be different (i.e. the type system used by the semantic analysis of your compiler would be different) and therefore your program could end up computing different results compared to the same program compiled by a real Java compiler that implements the typing rules from the Java reference manual.

To finish with type checkers, here is an interesting question: what is type checking, from the point of view of the Curry-Howard Isomorphism? Type checking is just proving little mathematical theorems about the source program: the theorems that say that all the different parts of the source program do not have type errors. In essence, the type checker inside a compiler (for a strongly typed programming language) is just a small, specialized theorem prover that automatically proves for you all these little theorems about the correctness of all the types in your source program. We know such theorems can be proved by the type checker because the type checker for a strongly typed programming language implements a type system which is equivalent to some mathematical logic in which all these little type theorems about your source program can be proven.

Type Compatibility

Let's look again at the typing rule for the `mod` operator:

$E_1 \rightarrow E_2$ `mod` E_3	if(E_2.type == integer and E_3.type == integer) then E_1.type := integer else E_1.type := type_error

One thing we have not explained yet in such a rule is the meaning of the type expression comparison operator "==". For basic types the meaning of == is clear enough: for example we expect one type expression integer to be == to another type expression integer, we expect one type expression boolean to be == to another type expression boolean, and we expect a type expression integer to not be == to a type expression boolean.

For constructed types things become more complicated: is the type expression pointer(integer) == to the type expression pointer(boolean)? After all both type expressions are pointers... In practice this is acceptable in a weakly typed, unsafe programming language like C (where booleans and integers are the same thing) but in a strongly typed, safe programming language like Java we do not want these two type expressions to be ==, because then it would be possible to transform an integer into a boolean and vice-versa: for a given integer, take the address (i.e. a pointer) of that integer, transform that pointer to integer into a pointer to boolean (assuming pointer(integer) and pointer(boolean) are ==), then dereference the pointer to boolean, which means your program would end up reading the memory bits of the integer as if these bits represented a boolean. This would obviously break the safety of the programming language (suddenly you would be ale to add booleans, or use integers are the test expression of an `if` statement) which would completely change the meaning of the programming language and remove any guarantee that Java gives about always detecting all type errors at compile time (except for downcasts between classes, which we are not looking at here) and always detecting all other errors at runtime.

So for a programming language like Java, we want the two type expressions pointer(integer) and pointer(boolean) to not be ==. To do this we not only have to check the fact that these two type expressions are both pointer type expressions, but we also have to check that both pointer type expressions are for pointers pointing to the exact same kinds of values, i.e. we also need to compare integer with boolean. This means that, when we have constructed type expressions like pointer(integer) and pointer(boolean) we need not only to compare the outer type constructor pointer, but we then also need to recursively check the type sub-expressions inside the pointer type constructors. To do this, we use the following typeequiv function, which implements the == operator inside the type checker of a compiler:

```
bool typeequiv(s, t){
    if(s and t are the same basic type expression)
        return true;
    if(s is pointer(s1) and t is pointer(t1))
        return typeequiv(s1, t1);
    if(s is s1 → s2 and t is t1 → t2)
        return (typeequiv(s1, t1) and typeequiv(s2, t2));
    if(s is array(is, s1) and t is array(it, t1))
        return (setequiv(is, it) and typeequiv(s1, t1));
    ... similarly for other type expression constructors ...
    return false;
}
```

For example, if we use this typeequiv function to compare pointer(integer) and pointer(boolean): first we call the function with these two type expressions as argument (called s and t). Neither s not t is a basic type expression (since they both use the pointer type expression constructor) so the first "if" test in the function is false and we move on the second one. In the second test, both s and t turn out to be using the pointer constructor, so the second test is true, with s_1 being integer and t_1 being boolean. The value returned by the function is then the same value returned as the result of recursively calling the same function on the two type sub-expressions integer and boolean. Inside that recursive call, both s and t are basic type expressions (integer and boolean) but they are not the same type expression (integer is different from boolean) so the first test inside the recursive call fails. Since s and t are basic types, all the other tests in the recursive call fail too, so the function reaches the last return statement and returns false, which then also becomes the result of the original call to typeequiv.

Similarly, if we use this function to compare pointer(integer) and pointer(integer), the first test will fail, the second test (for the pointer constructors) will succeed, the function will call itself recursively on the type sub-expression integer and integer, the first test inside the recursive call will then succeed (sine integer and integer are the same basic type expression) and the result will then be true.

The same idea applies to comparing type expressions for functions, except that we have then to compare recursively both the domains and ranges of the two function type expressions.

The case for arrays is a bit more interesting. If we have two array type expressions, then we compare the types of the elements of the two arrays using a recursive function call to the same function typeequiv. We also need to compare the sets of indexes of the two arrays. To do this we use a separate setequiv function, which implements the usual mathematical equality operation for sets of integers. So, for example, array(0...99, integer) and array(0...99, boolean) will not be equivalent because these two array type expressions do not describes the types of arrays that contain the same kind of elements. As another example, array(0...99, integer) and array(0...199, integer) will also not be equivalent, since the numbers of elements in the two arrays are different.

This last example in fact illustrates a problem with our typeequiv function. Imagine you have a sorting function that takes an array of 100 integers as formal argument:

```
sort(int[100] some_array){ ... }
```

Remember that the typing rule for function application requires that the type of the actual argument be == to the type of the formal argument of the function. This means that our `sort` function will work fine when given an array of exactly 100 integers:

```
int[100] my_array; ... sort(my_array) ...
```

but that the compiler will detect a type error when we try to give an array of 200 integers to the same function:

```
int[200] my_array; ... sort(my_array) ...
```

While in theory there is no problem with that, in practice this is very annoying, because it means we will need to implement a different `sort` function for every possible array size: one function to sort arrays of size 100, one function to sort arras of size 200, one function to sort arrays of size 53, etc. All these function will be exactly the same (same code) except for the size of the arrays that they manipulate. Needless to say, having hundreds of different functions that do all exactly the same thing except for very slightly different types is not good software engineering. In fact this was a huge problem in the Pascal programming language, because that language's equivalence function for type expressions really did require that the sets of indexes always be the same. This is one of the reasons why Pascal disappeared (some implementation in fact used modified typing rules that did not require the sets of indexes to be the same to go around this limitation, but these modified typing rules never became standard).

So in practice programming languages like C or Java (and most other programming languages) use a slightly different typeequiv function which does not compare the sets of indexes of arrays, and therefore does not call the setequiv function:

```
bool typeequiv(s, t){
    ...
    if(s is array(i_s, s_1) and t is array(i_t, t_1))
        return typeequiv(s_1, t_1);
    ...
}
```

This means that in C the following code compiles without type error:

```
sort(int[100] some_array){ ... } ... int[50] my_array; ... sort(my_array) ...
```

(except that in C you have to write "`int some_array[100]`" instead of "`int[100] some_array`").

Using the recursive typeequiv function above to implement the == comparison operator for type expression is what is called using *structural equivalence*: two type expressions are considered equal if they have the same structure, with the same type constructors (like pointer or array), and the same type subexpressions, at all recursive levels. Structural equivalence is used everywhere in all the typing rules of the C programming language, for example (with one exception, see below).

Structural equivalence is also used in Java for all the C-like data structures in that language: all the basic types, arrays, etc. (but not pointers since Java does not have an explicit notion of pointers). For classes though, Java uses another form of type expression equivalence called name equivalence, in which two classes are considered to be the same if they have the same name. For example, consider two classes `A` and `B` which have the exact same code:

```
class A { ... } ... class B { ... same code as A ... }
```

Even though these two classes have the exact same code (i.e. the corresponding objects have the exact same structure) the two classes are not considered equivalent, since they do not have the same name. Of course, in Java, name equivalence has to take inheritance into account:

```
class C extends A { ... some code ...}
```

In this example, even though class `C` does not have the same name as class `A`, and even though class `C` does not even have the same code as class `A` (i.e. the code of class `C` might override some methods of class `A`, or add new methods), an object of class `C` can still always be used as an object of class `A` (through an implicit upcast) since class `C` inherits from class `A`.

Name equivalence is in fact not limited to classes. In general *name equivalence* simply compares the names given to type expressions to decide whether they are equivalent or not, without looking at the internal structure of these type expressions. This is nice because there is no recursion involved, only direct name comparison, so it's much easier to implement.

For example, the C programming language uses name equivalence for structures (while it uses structural equivalence for everything else). This means that in C the following two structure definitions are not considered equivalent, even though internally they have the same number of members with the same types:

```
struct foo {int i; char c;}
struct bar {int i; char c;}
```

These two structure definitions are not equivalent since they do not use the same names (`foo` and `bar`). Of course in C nothing prevents you from using an explicit cast to forcibly convert one structure type into another, since C is weakly typed and unsafe, but by default the C type checker will not consider the two structure definitions to be the same.

The reason why the C language uses name equivalence for structure instead of structural equivalence is because structure types can contain loops:

```
struct list {int element; struct list * next;}
```

This structure definition defines a `list` structure which represents a single node in a linked list. Each node contains a single integer element as well as a pointer to the next node in the linked list. The type expression associated with such a structure is a record type expression: `list` = record(`element` $\times$ integer, `next` $\times$ pointer(`list`)) which is self-referential.

If we want to use the typeequiv function on two `list`-like type expressions, then we first have to extend the function to support records:

```
bool typeequiv(s, t){
    ...
    if(s is record(name_1^s × s_1, ..., name_i^s × s_i, ...)
    and t is record(name_1^t × t_1, ..., name_i^t × t_i, ...))
        return (typeequiv(s_1, t_1) and ... and typeequiv(s_i, t_i) and ...);
    ...
}
```

To compare the record type expressions s and t, this extended function recursively compares together the type sub-expressions s_i and t_i of matching members in the two records, based on the order in which they appear in the records (the names of the members do not matter when doing structural equivalence).

The problem is that, if we use this expended typeequiv function with two type expressions of the `list` kind above, then the function will go into an infinite recursive loop. To see this, let's consider two `list`-like type expressions with the same internal structures: $\texttt{list}_1$ = record(`element` $\times$ integer, `next` $\times$ pointer($\texttt{list}_1$)) and $\texttt{list}_2$ = record(`element` $\times$ integer, `next` $\times$ pointer($\texttt{list}_2$)). If we call the typeequiv function with $\texttt{list}_1$ and $\texttt{list}_2$ as arguments, the function will check that both type expressions are records and then will try to recursively compare the type sub-expressions for the members of the structures. So the typeequiv function will call itself recursively with the types integer and integer coming from the first member of $\texttt{list}_1$ and $\texttt{list}_2$ respectively, and the result will obviously be true. The function typeequiv will then call itself recursively with the types pointer($\texttt{list}_1$) and pointer($\texttt{list}_2$) coming from the second member of $\texttt{list}_1$ and $\texttt{list}_2$ respectively. Since both these two type expressions are pointer type expressions, the function will then call itself recursively again to check whether the type sub-expressions inside the pointer type expressions match or not. So typeequiv will call itself again this time with $\texttt{list}_1$ and $\texttt{list}_2$ as arguments. But this is exactly what we were trying to do in the first place with the original call to the typeequiv function! The result is that the typeequiv function will end up calling itself recursively for ever (or at least until the program runs out of stack space...) The whole problem is that the type expressions $\texttt{list}_1$ and $\texttt{list}_2$ refer to themselves, so trying to use structural equivalence with such data structures will result in an infinite recursive loop. To avoid this problem, the C programming language simply mandates that comparing structures should be done using name equivalence, not using structural equivalence.

Type Conversions

To finish with types, type expressions, type systems, and type checking, we have to look at type conversions. A type conversion is an operation that changes the type of an expression. For example, in C we can write the following piece of code:

```
float f; int i; ... f + i; ...
```

This works fine from the point of view of types, since there is no problem in mathematics with adding an integer with a real (floating point) number. There is a practical problem though: microprocessors have hardware for integer additions (adding an integer to an integer) and hardware for floating point additions (adding a floating point number to a floating point number) but no hardware for mixed additions (adding an integer to a floating point number or adding a floating point number to an integer). Microprocessor manufacturers (like Intel, or AMD) could add new hardware to their microprocessors to handle such mixed additions, but it would add a lot of complexity (i.e. cost more and probably reduce the overall speed of the microprocessor) for little gain (since, as we are going to see, the compiler can solve the problem itself) so in practice manufacturers are not interested in adding such new hardware.

We then have two solutions to solve this problem.

- Reject the program with a type error, telling the user that mixed additions are not allowed. The user could then edit his source program and manually add an explicit type cast to his code to transform the integer into a floating point number (since the floating point type is the one that represents the bigger set of possible runtime values):

  ```
  float f; int i; ... f + (float) i; ...
  ```

  This explicitly tells the compiler to insert extra code into the generated program to first convert the integer into a floating point number before doing a floating point addition.

- Let the compiler insert the type cast automatically for us. The type cast in then implicit in the source code (i.e. it is not visible in the source program but the semantics of the programming language mandates that the compiler always introduces such type cast, so we can say that the cast is "virtually" there in the source code...) As we are going to see, this is easy to do for the compiler, and is quite convenient for the user (a little more convenient than having to manually write an explicit cast), so many programming languages (but not all, OCaml being a well known exception) require that compiler insert such implicit casts in the program on behalf of the user.

To see how a compiler detects mixed additions and how it solves the problem, we first need to have in our programming language both integers and floating point numbers. We already have integers (represented by the `num` token):

$$E \to \texttt{num} \qquad E.\text{type} := \texttt{num}.\text{type}$$

where num.type is an intrinsic attribute which is always integer.

Similarly, we introduce floating point literal (constants), represented by the token float_lit:

$$E \to \texttt{float\_lit} \qquad E.\text{type} := \texttt{float\_lit}.\text{type}$$

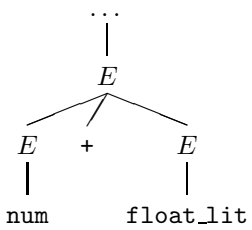
and the attribute float_lit.type is an intrinsic attribute which is defined one for all in the compiler to always be the type expression floatingpoint.

We then add an addition operator that is going to work with both integer and floating point numbers (unlike the mod operator that we have seen before, which only works with integers). To detect mixed additions we then simply have to look at the type expressions for the addition's operands:

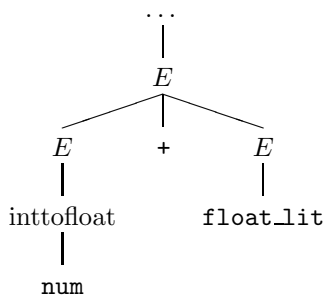
$E_1 \to E_2$ + E_3

if(E_2.type == integer and E_3.type == integer)
then E_1.type := integer
if(E_2.type == floatingpoint and E_3.type == floatingpoint)
then E_1.type := floatingpoint
if(E_2.type == integer and E_3.type == floatingpoint)
then { insert_type_conversion(E_2, inttofloat)
       E_1.type := floatingpoint
if(E_2.type == floatingpoint and E_3.type == integer)
then { insert_type_conversion(E_3, inttofloat)
       E_1.type := floatingpoint
else E_1.type := type_error

So, if we are adding two integers, then the result is an integer. If we are adding two floating point numbers, then the result is a floating point number. If we are adding an integer to a floating point number, then the result is again a floating point number, but this time we also have to insert in the parse tree for the addition a type conversion from integer to floating point number (called "inttofloat") under the nonterminal E_2. This is what the procedure insert_type_conversion does: given a nonterminal like E_2 in a parse tree and a type conversion like inttofloat, it inserts the type conversion just below the nonterminal E_2 in the parse tree. So, for example, if the original parse tree looked like this:



then after calling the procedure insert_type_conversion, the parse tree will now look like this:



with the extra type conversion inttofloat inserted just below E_2. Then both operands of the addition are floating point numbers, and then in the code generation phase of the compiler (during the instruction selection, to be precise) we will have no problem finding a microprocessor instruction to implement the addition: we will just use the microprocessor's floating point addition instruction.

If we are adding a floating point number to an integer, then the result is again a floating point number, but this time we have to insert the inttofloat type conversion under E_3, not E_2. Finally, if one (or both) of the operands is neither an integer nor a floating point number then we just have a type error.

Of course the same technique works not just for addition but for all other arithmetic operators (subtraction, multiplication, division) that support both integer and floating point operations at the hardware level but do not support operations with mixed types. So we can generalize our typing rule and replace the `+` operator with a more general `arith_op` token that represents all such arithmetic operators.