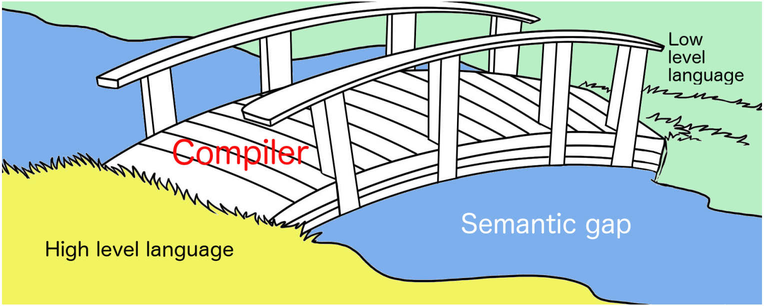
Compilers and computer architecture: Semantic analysis

Martin Berger [1]

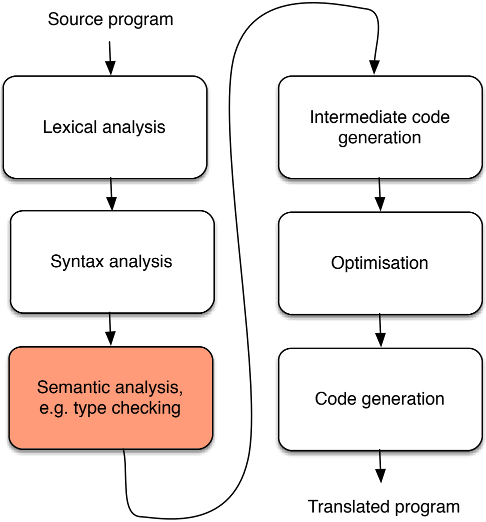
October / November 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall the structure of compilers



Semantic analysis

One of the jobs of the compiler front-end is to reject ill-formed inputs. This is usually done in three stages.

- Lexical analysis: detects inputs with illegal lexical syntax.
- Parsing: detects inputs with ill-formed syntax (no parse-tree).
- Semantic analysis: catch 'all' remaining errors, e.g. variable used before declared. 'Last line of defense'.

Semantic analysis

One of the jobs of the compiler front-end is to reject ill-formed inputs. This is usually done in three stages.

- Lexical analysis: detects inputs with illegal lexical syntax.
- Parsing: detects inputs with ill-formed syntax (no parse-tree).
- Semantic analysis: catch 'all' remaining errors, e.g. variable used before declared. 'Last line of defense'.

Why do we need a separate semantic analysis phase at all?

Semantic analysis

One of the jobs of the compiler front-end is to reject ill-formed inputs. This is usually done in three stages.

- Lexical analysis: detects inputs with illegal lexical syntax.
- Parsing: detects inputs with ill-formed syntax (no parse-tree).
- Semantic analysis: catch ’all’ remaining errors, e.g. variable used before declared. ’Last line of defense’.

Why do we need a separate semantic analysis phase at all?

Answer: Some language constraints are not expressible using CFGs (too complicated).

The situation is similar to the split between lexing and parsing: not everything about syntactic well-formedness can be expressed by regular expressions & FSAs, so we use CFGs later.

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?
- Are classes and variables defined only once?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?
- Are classes and variables defined only once?
- Methods defined only once?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?
- Are classes and variables defined only once?
- Methods defined only once?
- Are private methods and members only used within the defining class?

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?
- Are classes and variables defined only once?
- Methods defined only once?
- Are private methods and members only used within the defining class?
- Stupid operations like $cosine(true)$ or $"hello"/7$?.

Caveat

When we say that semantic analysis catches ’all’ remaining errors, that does not include application-specific errors. It means catching errors that violate the well-formedness constraints that the language iteself imposes.

Naturally, those constraints are chosen by the language designers with a view towards efficient checkability by the compiler.

Rice's Theorem and undecidability

Rice's theorem. No interesting property of programs (more precisely: program execution) is decidable.

Rice's Theorem and undecidability

Rice's theorem. No interesting property of programs (more precisely: program execution) is decidable.

That means for essentially any property that programs might have (e.g. does not crash, terminates, loops forever, uses more than 1782349 Bytes of memory) there cannot be a perfect checker, ie. a program that determines with **perfect accuracy** whether the chosen property holds of any input program or not.

Informally, one may summarise Rice's theorem as follows: **to work out with 100% certainty what programs do, you have to run them (with the possibility of non-termination), there is no shortcut**.

Rice's Theorem and undecidability

Not all hope is lost!

Rice's Theorem and undecidability

Not all hope is lost!

We can **approximate** a property of interest, and our approximation will have either false positives or false negatives (or both).

Rice's Theorem and undecidability

So our semantic analysis must approximate. A compiler does this in a **conservative** way ("erring on the side of caution"): every program the semantic analysis accepts is guaranteed to have to properties that the semantic analysis check for, but the semantic analysis will reject a lot of safe programs (having the required property).

Rice's Theorem and undecidability

So our semantic analysis must approximate. A compiler does this in a **conservative** way ("erring on the side of caution"): every program the semantic analysis accepts is guaranteed to have to properties that the semantic analysis check for, but the semantic analysis will reject a lot of safe programs (having the required property).

Example: Our semantic analysis guarantees that programs never try to multiply an integer and a string like `cosine("hello")`. In this sense, the following program is safe (why?):

Rice's Theorem and undecidability

So our semantic analysis must approximate. A compiler does this in a **conservative** way ("erring on the side of caution"): every program the semantic analysis accepts is guaranteed to have to properties that the semantic analysis check for, but the semantic analysis will reject a lot of safe programs (having the required property).

Example: Our semantic analysis guarantees that programs never try to multiply an integer and a string like `cosine("hello")`. In this sense, the following program is safe (why?):

```
if ( x*x = -1 ) {
   y = 3 / "hello" }
else
   y = 3 / 43110 }
```

Yet any typing system in practical use will reject it. (Why?)

Plan

For lexing and parsing we proceeded in two steps.

1. Specify constraint (RE for lexing, CFGs for parsing)
2. Invented algorithm to check constraints given in (1): FSA to decide REs, (top-down) parser to decide CFGs.

Plan

For lexing and parsing we proceeded in two steps.

1. Specify constraint (RE for lexing, CFGs for parsing)
2. Invented algorithm to check constraints given in (1): FSA to decide REs, (top-down) parser to decide CFGs.

For semantic analysis such a nice separation between specification and algorithm is difficult / an open problem. It seems hard to express constraints independent from giving an algorithm that checks for them.

The whole session on semantic analysis will be more superficial than those on lexing/parsing.

Semantic analysis by traversal of the AST

Just like code generation, semantic analysis will happen by walking the AST:

Semantic analysis by traversal of the AST

Just like code generation, semantic analysis will happen by walking the AST:

- Analyses a node N in the AST.

Semantic analysis by traversal of the AST

Just like code generation, semantic analysis will happen by walking the AST:

- Analyses a node *N* in the AST.
- Recursively process the children of *N*.

Semantic analysis by traversal of the AST

Just like code generation, semantic analysis will happen by walking the AST:

- Analyses a node N in the AST.
- Recursively process the children of N.
- After the recursion terminates, finish the analysis of N with the information obtained from the children.

Semantic analysis by traversal of the AST

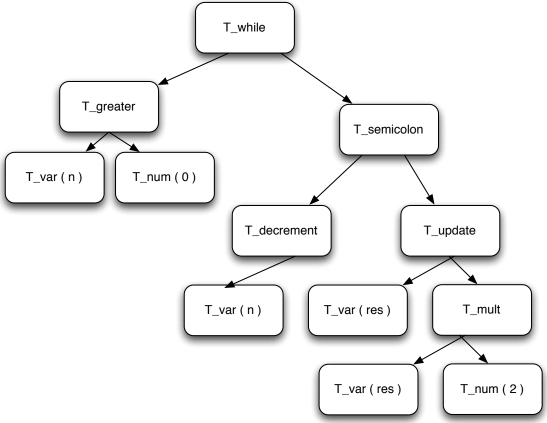
Semantic analysis by traversal of the AST

Semantic analysis by traversal of the AST

```
while( n > 0 ){
     n--;
     res *= 2; }
```

Semantic analysis by traversal of the AST

```
while( n > 0 ){
     n--;
     res *= 2; }
```



Semantic analysis by traversal of the AST

Sometimes we will have to walk the AST multiple times (e.g. some kinds of type-inference).

Semantic analysis by traversal of the AST

Sometimes we will have to walk the AST multiple times (e.g. some kinds of type-inference).

We want to be sure that every identifier is declared. With non-recursive definitions this is no problem, everything is declared before use, which is easy to check with one recursive descent of the AST.

Semantic analysis by traversal of the AST

Sometimes we will have to walk the AST multiple times (e.g. some kinds of type-inference).

We want to be sure that every identifier is declared. With non-recursive definitions this is no problem, everything is declared before use, which is easy to check with one recursive descent of the AST.

With recursive definitions sometimes identifiers are declared after use. What do we do?

Semantic analysis by traversal of the AST

Sometimes we will have to walk the AST multiple times (e.g. some kinds of type-inference). With recursive definitions sometimes identifiers are declared after use. What do we do?

```
abstract class A () {
     B b  }

abstract class B () {
     A a }
```

Semantic analysis by traversal of the AST

Sometimes we will have to walk the AST multiple times (e.g. some kinds of type-inference). With recursive definitions sometimes identifiers are declared after use. What do we do?

```
abstract class A () {
     B b  }

abstract class B () {
     A a }
```

Answer:

- Walk the AST once collecting class definitions.
- Walk the AST a second time checking if each use of a class (type) identifier has a definition somewhere.
- Alternatively, propagate information about needed definitions up in a clever way and check if everything is OK.

Key tool: types

Types and typing systems are the key tool for semantic analysis.

Key tool: types

Types and typing systems are the key tool for semantic analysis.

What are types?

Key tool: types

Types and typing systems are the key tool for semantic analysis.

What are types?

Types are a **rough classification** of programs that rule out certain errors, e.g. `cosine( "hello")`!

Key tool: types

Types and typing systems are the key tool for semantic analysis.

What are types?

Types are a **rough classification** of programs that rule out certain errors, e.g. `cosine( "hello")`!

With each type *t* we associate values, and operators that we can apply to values of type *t*. Conversely, with each operator with associate types that describe the nature of the operator's arguments and result.

For example to values of type `string`, we can apply operations such as `println`, but we cannot multiply two strings.

Types

In mainstream PLs, types are weak (= not saying anything complicated) specifications of programs.

Types as two-version programming

Types as two-version programming

In languages such as Java, programs are annotated with types. This can be seen as a weak form of two-version programming: the programmer specifies twice what the program should do, once by the actual code, and a second time through the types.

By saying something twice, but in somewhat different languages (Java vs types) the probability that we make the same mistake in both expressions is lower than if we state our intention only once.

Types as two-version programming

In languages such as Java, programs are annotated with types. This can be seen as a weak form of two-version programming: the programmer specifies twice what the program should do, once by the actual code, and a second time through the types.

By saying something twice, but in somewhat different languages (Java vs types) the probability that we make the same mistake in both expressions is lower than if we state our intention only once.

The key idea behind semantic analysis is to look for **contradictions** between the two specifications, and reject programs with such contradictions.

Types

Note that types can only prevent stupid mistakes like `"hello" * "world"`.

They cannot (usually) prevent more complicated problems, like out-of-bounds indexing of arrays.

```
int [] a = new int [ 10 ]
a [ 20 ] = 3
```

Type-checking vs type-inference

An important distinction is that between **type-checking** (old-fashioned) and **type-inference** (modern).

Type-checking vs type-inference

An important distinction is that between **type-checking** (old-fashioned) and **type-inference** (modern).

- In type-checking (e.g. Java) we verify that the programmer-written type-annotations are consistent with the program code. E.g.

  ```
  def f ( x : String ) : Int = {
    if ( x = "Moon" )
      true
    else
      false }
  ```

  is easy to see as inconsistent.

Type-checking vs type-inference

An important distinction is that between **type-checking** (old-fashioned) and **type-inference** (modern).

- In type-checking (e.g. Java) we verify that the programmer-written type-annotations are consistent with the program code. E.g.

  ```
  def f ( x : String ) : Int = {
    if ( x = "Moon" )
      true
    else
      false }
  ```

  is easy to see as inconsistent.
- In type-inference (e.g. Haskell, Ocaml, Scala, Rust, ...) we analyse the program code to see if it is **internally consistent**. This is done by trying to find types that we could assign to a program to make it type-check. (So we let the computer do most of the work of type annotations.)

Type-checking vs type-inference summary

Type-checking: is the program consistent with programmer-supplied type annotations?

Type-inference: is the program consistent with itself?

Type-checking vs type-inference

```
def f ( y : ??? ) : ??? = {
  if ( x = y )
    y
  else
    x+1 }
```

What types could you give to x, y and the return value of f?

Type-checking vs type-inference

```
def f ( y : ??? ) : ??? = {
  if ( x = y )
    y
  else
    x+1 }
```

What types could you give to *x*, *y* and the return value of *f*?

Clearly *x* has type integer, *y* has type integer.

Type-checking vs type-inference

That was easy. What about this program

```
def f ( x : ??? ) : ??? = {
  while ( x.g ( y ) ) {
    y = y+1 };
  if ( y > z )
    z = z+1
  else
    println ( "hello" ) }
```

What types could you give to x, y, z, g and f?

Type-checking vs type-inference

That was easy. What about this program

```
def f ( x : ??? ) : ??? = {
  while ( x.g ( y ) ) {
    y = y+1 };
  if ( y > z )
    z = z+1
  else
    println ( "hello" ) }
```

What types could you give to x, y, z, g and f?

y and z are integers, x must be a class A such that A has a method (should it be public or private?) g which takes and integer and returns a boolean.
Finally, f returns nothing, so should be of type *Unit*, or *void* (which is the same thing in many contexts).

Polymorphism (not exam relevant)

What about this program?

```
def f ( x : ??? ) : ??? = { return x }
```

Polymorphism (not exam relevant)

What about this program?

```
def f ( x : ??? ) : ??? = { return x }
```

We can use **any** type *t*, as long as the input and output both have *t*:

```
def f ( x : t ) : t = { return x }
```

This is called **(parametric) polymorphism.**

Polymorphism (not exam relevant)

But which concrete type *t* should we use for

```
def f ( x : t ) : t = { return x }
```

We want to be able to do `f(17)` **and** `f(true)` in the same program. Any concrete choice of *t* would prevent this. In order to deal with this we assign a **type variable** which can be instantiated with any type.

Polymorphism (not exam relevant)

But which concrete type *t* should we use for

```
def f ( x : t ) : t = { return x }
```

We want to be able to do `f(17)` **and** `f(true)` in the same program. Any concrete choice of *t* would prevent this. In order to deal with this we assign a **type variable** which can be instantiated with any type.

In Java, the concept of **generics** captures this polymorphism, e.g.:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();}
```

Advanced question (not exam relevant)

What types should/could we infer for e.g.

```
def f ( x : ??? ) : ??? = {
   if ( x > 3.1415 ) then
        true
   else
       throw new Exception ( "...!" ) }
```

Dynamic vs static typing

An important distinction is that between **dynamically typed languages**, e.g. Python, Javascript, and **statically typed languages** such as Java, C, C++. The difference is **when** type-checking happens.

- In dynamically typed languages, type-checking happens at run-time, at the last possible moment, e.g. just before we execute `x+2` we check that x contains an integer.
- In statically typed languages, type-checking happens at compile-time

Dynamic vs static typing

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantage for dynamically typed languages: more flexibility. There are many programs that are safe, but cannot be typed at compile time, e.g.

```
x = 100
x = x*x
println ( x+1 )
x = "hello"
x = concatenate( x, " world" )
println ( x )
```

Moreover, vis-a-vis languages like Java, we don't have to write type annotations. Less work ...

Dynamic vs static typing

The compilation of dynamically typed languages (and how to make them fast) using JIT compilers is very interesting and a hot research topic.

Type inference for advanced programming languages is also very interesting and a hot research topic.

However, from now on we will only look at statically typed languages with type-checking (except maybe later in the advanced topics).

Dynamic vs static typing

For large-scala industrial programming, the disadvantages of dynamically typed languages become overwhelming, and static types are often retro-fitted, e.g.:

- Javascript: Flow (Facebook) and Typescript (Microsoft)
- Python: mypy and PyAnnotate (Dropbox)

Type-checking for a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

$$
\begin{array}{rcl}
P & ::= & x \mid 0 \mid 1 \mid ... \mid \texttt{true} \mid \texttt{false} \mid P = P \mid P < P \mid \\
 & \mid & P + P \mid P * P \mid \texttt{for}\ i\ =\ P\ \texttt{to}\ P\ \texttt{do}\ \{P\} \mid \texttt{new List<}\alpha\texttt{>} \\
 & \mid & P\texttt{.append(}P\texttt{)} \mid P\texttt{.get(}P\texttt{)} \\
 & \mid & \texttt{while}\ P\ \texttt{do}\ \{P\} \mid \texttt{if}\ P\ \texttt{then}\ P\ \texttt{else}\ P \\
 & \mid & x := P \mid \texttt{let x :}\ \alpha = \texttt{P in P} \mid P; P
\end{array}
$$

Here x ranges over variables and α over types (see below).

Type-checking for a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

$$
\begin{array}{lcl}
P & ::= & x \mid 0 \mid 1 \mid \dots \mid \texttt{true} \mid \texttt{false} \mid P = P \mid P < P \mid \\
 & \mid & P + P \mid P * P \mid \texttt{for}\ i\ =\ P\ \texttt{to}\ P\ \texttt{do}\ \{P\} \mid \texttt{new List<}\alpha\texttt{>} \\
 & \mid & P\texttt{.append(}P\texttt{)} \mid P\texttt{.get(}P\texttt{)} \\
 & \mid & \texttt{while}\ P\ \texttt{do}\ \{P\} \mid \texttt{if}\ P\ \texttt{then}\ P\ \texttt{else}\ P \\
 & \mid & x := P \mid \texttt{let x} : \alpha = \texttt{P in P} \mid P; P
\end{array}
$$

Here x ranges over variables and α over types (see below).

We want to do semantic analysis that catches mistakes like $\texttt{true} + 7$ and we use types for this purpose.

Type-checking for a simple imperative language

Now we need to define types. Here they are.

$$\alpha \quad ::= \quad \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{List<}\alpha\texttt{>}$$

The type `Unit` is the type of statements (like `void` in Java). Clearly `Int` is the type of integers, and `Bool` that of booleans. Finally $\texttt{List<}\alpha\texttt{>}$ is the type of lists storing things of type α.

Type-checking for a simple imperative language

Now we need to define types. Here they are.

$$\alpha \quad ::= \quad \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{List<}\alpha\texttt{>}$$

The type `Unit` is the type of statements (like `void` in Java). Clearly `Int` is the type of integers, and `Bool` that of booleans. Finally `List<`α`>` is the type of lists storing things of type α.

The Java program

```
int x = 3;
x = x+1;
```

in our language is

$$\texttt{let}\ \texttt{x} : \texttt{Int} = \texttt{3}\ \texttt{in}\ \texttt{x} := \texttt{x} + \texttt{1}$$

Type-checking for a simple imperative language

A Java program like

```
List<Int> a = new List<Int>();
a.append(10);
```

translates to

```
let a : List<Int> = new List<Int> in a.append(10)
```

Type-checking for a simple imperative language

Let's look at some examples.

Type-checking for a simple imperative language

Let's look at some examples.

- The program 3 has type `Int`

Type-checking for a simple imperative language

Let's look at some examples.

- The program 3 has type `Int`
- The program `true` has type `Bool`

Type-checking for a simple imperative language

Let's look at some examples.

- The program 3 has type `Int`
- The program `true` has type `Bool`
- What about $3 + 4$?

Type-checking for a simple imperative language

Let's look at some examples.

- The program 3 has type `Int`
- The program `true` has type `Bool`
- What about $3 + 4$?
- What about $3 + x$?

Type-checking for a simple imperative language

The type of programs like $3 + x$ depends on our assumptions about the type of x: if we assume that x has type `Int` then $3 + x$ has type `Int`.

Type-checking for a simple imperative language

The type of programs like $3 + x$ depends on our assumptions about the type of x: if we assume that x has type `Int` then $3 + x$ has type `Int`.

If we assume that x has type `Bool` then $3 + x$ has no type!

Type-checking for a simple imperative language

The type of programs like $3 + x$ depends on our assumptions about the type of x: if we assume that x has type `Int` then $3 + x$ has type `Int`.

If we assume that x has type `Bool` then $3 + x$ has no type!

An executable program has no free variables, unlike $3 + x$, since **all variables have to be declared before use.** (Why?)

Type-checking for a simple imperative language

If all variables have to be declared before use, why do we have to worry about programs with free variables at all when the programs we run don't have free variables?

Type-checking for a simple imperative language

We want to type-check in a **compositional** way, that means, determining types of a program from the types of its components.

The key construct here is

$$\texttt{let x} : \alpha = \texttt{P in Q}.$$

To type-check *Q* we have to add the assumption that *x* **stores** something of type α to the assumptions we use to type-check the whole phrase $\texttt{let x} : \alpha = \texttt{P in Q}$.

Type-checking for a simple imperative language

We want to type-check in a **compositional** way, that means, determining types of a program from the types of its components.

The key construct here is

$$\texttt{let x} : \alpha = \texttt{P in Q}.$$

To type-check *Q* we have to add the assumption that *x* **stores** something of type α to the assumptions we use to type-check the whole phrase $\texttt{let x} : \alpha = \texttt{P in Q}$.

Assume *y* stores something of type `Int`. Under this assumption, the program

$$\texttt{let x} : \texttt{Int} = \texttt{y} + \texttt{1 in x} := \texttt{x} + \texttt{2}$$

is well-typed and has type `Unit`.

Type-checking for a simple imperative language

In other words, we type-check using **assumptions** about the types of free variables. We can split this insight into parts (divide-and-conquer):

Type-checking for a simple imperative language

In other words, we type-check using **assumptions** about the types of free variables. We can split this insight into parts (divide-and-conquer):

- We need to store the assumptions.

Type-checking for a simple imperative language

In other words, we type-check using **assumptions** about the types of free variables. We can split this insight into parts (divide-and-conquer):

- We need to store the assumptions.
- We need to be able to get the assumptions.

Type-checking for a simple imperative language

In other words, we type-check using **assumptions** about the types of free variables. We can split this insight into parts (divide-and-conquer):

- We need to store the assumptions.
- We need to be able to get the assumptions.

So our type-checking algorithm needs a suitable data structure, an 'assumption store'.

Type-checking for a simple imperative language

To describe the type-checking algorithm concisely, let's introduce some notation.

We write $\Gamma \vdash P : \alpha$, meaning that program P has type α under the **assumptions** as given by Γ. This Γ is our 'assumption store', and pronounced "Gamma". The 'assumption store' is also called **symbol table** or **typing environment** or just **environment**. You can think of Γ as a function: you pass a variable name to Γ and get either an error (if Γ has no assumptions on that variable) or a type α (if Γ stores the assumption that the variable has type α).

Type-checking for a simple imperative language

To describe the type-checking algorithm concisely, let's introduce some notation.

We write $\Gamma \vdash P : \alpha$, meaning that program P has type α under the **assumptions** as given by Γ. This Γ is our 'assumption store', and pronounced "Gamma". The 'assumption store' is also called **symbol table** or **typing environment** or just **environment**. You can think of Γ as a function: you pass a variable name to Γ and get either an error (if Γ has no assumptions on that variable) or a type α (if Γ stores the assumption that the variable has type α).

We write $\Gamma(x) = \alpha$ to indicate that Γ stores the assumption that x has type α.

Type-checking for a simple imperative language

To describe the type-checking algorithm concisely, let's introduce some notation.

We write $\Gamma \vdash P : \alpha$, meaning that program P has type α under the **assumptions** as given by Γ. This Γ is our 'assumption store', and pronounced "Gamma". The 'assumption store' is also called **symbol table** or **typing environment** or just **environment**. You can think of Γ as a function: you pass a variable name to Γ and get either an error (if Γ has no assumptions on that variable) or a type α (if Γ stores the assumption that the variable has type α).

We write $\Gamma(x) = \alpha$ to indicate that Γ stores the assumption that x has type α.

We sometimes want to **add** and assumption $x : \alpha$ to the assumptions already in Γ. We write $\Gamma, x : \alpha$ in this case, assuming that Γ does not already store assumptions about x.

Type-checking for a simple imperative language

- $\Gamma \vdash \texttt{true} : \texttt{Bool}$

Type-checking for a simple imperative language

- $\Gamma \vdash \texttt{true} : \texttt{Bool}$
- $\Gamma \vdash \texttt{false} : \texttt{Bool}$

Type-checking for a simple imperative language

- $\Gamma \vdash \texttt{true} : \texttt{Bool}$
- $\Gamma \vdash \texttt{false} : \texttt{Bool}$
- $\Gamma \vdash 7 : \texttt{Int}$

These type can be derived without assumptions on the types of variables, and other program parts.

Type-checking for a simple imperative language

- If $\Gamma(x) = \alpha$ then $\Gamma \vdash x : \alpha$

Type-checking for a simple imperative language

- If $\Gamma(x) = \alpha$ then $\Gamma \vdash x : \alpha$
- If $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P + Q : \texttt{Int}$.

Type-checking for a simple imperative language

- If $\Gamma(x) = \alpha$ then $\Gamma \vdash x : \alpha$
- If $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P + Q : \texttt{Int}$.
- If $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P = Q : \texttt{Bool}$.

Type-checking for a simple imperative language

- If $\Gamma(x) = \alpha$ then $\Gamma \vdash x : \alpha$
- If $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P + Q : \texttt{Int}$.
- If $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P = Q : \texttt{Bool}$.

Writing out "if" etc explicitly gets unwieldy quickly, so let's introduce a new form of notation.

Type-checking for a simple imperative language

We write

$$\frac{\mathit{Assumption}_1 \quad \ldots \quad \mathit{Assumption}_n}{\mathit{Conclusion}}$$

for: whenever *Assumption*$_1$ and ... and *Assumption*$_n$ are true, then *Conclusion* is true.

Type-checking for a simple imperative language

We write

$$\frac{Assumption_1 \quad \ldots \quad Assumption_n}{Conclusion}$$

for: whenever $Assumption_1$ and ... and $Assumption_n$ are true, then *Conclusion* is true.

Example: we write

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P + Q : \texttt{Int}}$$

for:

Type-checking for a simple imperative language

We write

$$\frac{\textit{Assumption}_1 \quad \ldots \quad \textit{Assumption}_n}{\textit{Conclusion}}$$

for: whenever *Assumption*$_1$ and ... and *Assumption*$_n$ are true, then *Conclusion* is true.

Example: we write

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P + Q : \texttt{Int}}$$

for: if $\Gamma \vdash P : \texttt{Int}$ and also $\Gamma \vdash Q : \texttt{Int}$ then $\Gamma \vdash P + Q : \texttt{Int}$.

Type-checking for a simple imperative language

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash 7 : \texttt{Int}}$$

$$\frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha} \qquad \frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P + Q : \texttt{Int}}$$

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P = Q : \texttt{Bool}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash P : \texttt{Unit} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P; Q : \texttt{Unit}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash P : \texttt{Unit} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P; Q : \texttt{Unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Bool} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash \texttt{while}\ C\ \texttt{do}\ \{Q\} : \texttt{Unit}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash P : \texttt{Unit} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P; Q : \texttt{Unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Bool} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash \texttt{while}\ C\ \texttt{do}\ \{Q\} : \texttt{Unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Bool} \quad \Gamma \vdash Q : \alpha \quad \Gamma \vdash R : \alpha \quad \alpha \text{ arbitrary type}}{\Gamma \vdash \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R : \alpha}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash P : \texttt{Unit} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P; Q : \texttt{Unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Bool} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash \texttt{while}\ C\ \texttt{do}\ \{Q\} : \texttt{Unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Bool} \quad \Gamma \vdash Q : \alpha \quad \Gamma \vdash R : \alpha \quad \alpha \text{ arbitrary type}}{\Gamma \vdash \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R : \alpha}$$

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int} \quad i \text{ not def in } \Gamma \quad \Gamma, i : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for}\ i\ =\ P\ \texttt{to}\ Q\ \texttt{do}\ \{R\} : \texttt{Unit}}$$

Recall that $\Gamma, i : \texttt{Int}$ means that we add the assumption that i has type `Int` to our environment.

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

$$\frac{\Gamma \vdash P : \alpha \quad x \text{ not defined in } \Gamma \quad \Gamma, x : \alpha \vdash Q : \beta}{\Gamma \vdash \texttt{let x} : \alpha = \texttt{P in Q} : \beta}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

$$\frac{\Gamma \vdash P : \alpha \quad x \text{ not defined in } \Gamma \quad \Gamma, x : \alpha \vdash Q : \beta}{\Gamma \vdash \texttt{let x} : \alpha = \texttt{P in Q} : \beta}$$

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

$$\frac{\Gamma \vdash P : \alpha \quad x \text{ not defined in } \Gamma \quad \Gamma, x : \alpha \vdash Q : \beta}{\Gamma \vdash \texttt{let x} : \alpha = \texttt{P in Q} : \beta}$$

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \quad \Gamma \vdash Q : \alpha}{\Gamma \vdash P\texttt{.append(}Q\texttt{)} : \texttt{List<}\alpha\texttt{>}}$$

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

$$\frac{\Gamma \vdash P : \alpha \quad x \text{ not defined in } \Gamma \quad \Gamma, x : \alpha \vdash Q : \beta}{\Gamma \vdash \texttt{let x} : \alpha = \texttt{P in Q} : \beta}$$

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \quad \Gamma \vdash Q : \alpha}{\Gamma \vdash P\texttt{.append(}Q\texttt{)} : \texttt{List<}\alpha\texttt{>}}$$

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P\texttt{.get(}Q\texttt{)} : \alpha}$$

Alternatives?

Note that alternative rules are also meaningful, e.g.

$$\frac{\Gamma \vdash P : \alpha \quad \Gamma \vdash Q : \beta}{\Gamma \vdash P; Q : \beta} \qquad \frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash P : \alpha}{\Gamma \vdash x := P : \alpha}$$

Question: what is returned in both cases?

AST in pseudo-code

```
interface Prog
  class Ident ( s : String ) implements Prog
  class IntLiteral ( i : Int ) implements Prog
  class BoolLiteral ( b : Boolean ) implements Prog
  class Equal ( lhs : Prog, rhs : Prog ) implements Prog
  class Plus ( lhs : Prog, rhs : Prog ) implements Prog
  class For ( i : String,
              from : Prog,
              to : Prog, body : Prog ) implements Prog
  class While ( cond : Prog, body : Prog ) implements Pr
  class If ( cond : Prog, sThen : Prog, sElse : Prog ) i
  class Assign ( i : String, e : Prog ) implements Prog
  class Let ( x : String, t : Type, p : Prog, q : Prog )
  class NewList ( t: Type ) implements Prog
  class Append ( list: Prog, elem: Prog ) implements Pro
  class Get ( list: Prog, index: Prog ) implements Prog
```

AST in pseudo-code

```
interface Type
  class Int_T () implements Type
  class Bool_T () implements Type
  class Unit_T () implements Type
  class List_T ( ty : Type ) implements Type
```

Symbol-tables in (pseudo-) code

Remember: a key data structure in semantic analysis is the **symbol table**, or **environment**, which we wrote Γ above.

Symbol-tables in (pseudo-) code

Remember: a key data structure in semantic analysis is the **symbol table**, or **environment**, which we wrote Γ above.

The symbol table maps identifiers (names, variables) to their types (here we think of a class signature as the type of a class).

Symbol-tables in (pseudo-) code

Remember: a key data structure in semantic analysis is the **symbol table**, or **environment**, which we wrote Γ above.

The symbol table maps identifiers (names, variables) to their types (here we think of a class signature as the type of a class).

We use the symbol table to track the following.

- Is every used variable defined (exactly once)?
- Is every variable used according to its type? E.g. if x is declared to be a string, we should not try `x + 3`.

Symbol-tables in (pseudo-) code

In Java we could do this:

```
HashMap<String, Type> env =
        new HashMap<String, Type>();
```

Symbol-tables in (pseudo-) code

In Java we could do this:

```
HashMap<String, Type> env =
        new HashMap<String, Type>();

// Returns the type associated with x.
env.get(x)
```

Symbol-tables in (pseudo-) code

In Java we could do this:

```
HashMap<String, Type> env =
        new HashMap<String, Type>();

// Returns the type associated with x.
env.get(x)

// Adds the association of x with type t.
// Removes any previous association for x.
env.put(x, t)
```

Symbol-tables in (pseudo-) code

In Java we could do this:

```
HashMap<String, Type> env =
        new HashMap<String, Type>();

// Returns the type associated with x.
env.get(x)

// Adds the association of x with type t.
// Removes any previous association for x.
env.put(x, t)

// Returns true if there exists an
// association for x.
env.containsKey(x)
```

Symbol-tables in (pseudo-) code

In Java we could do this:

```
HashMap<String, Type> env =
        new HashMap<String, Type>();

// Returns the type associated with x.
env.get(x)

// Adds the association of x with type t.
// Removes any previous association for x.
env.put(x, t)

// Returns true if there exists an
// association for x.
env.containsKey(x)

// Returns // association for x, if it exists.
env.remove(x)
```

Type-checking for a simple imperative language

```
env.put ( x, Int_T )
println ( env.get ( x ) // prints Int_T
env.put ( x, Bool_T )
println ( env.get ( x ) // prints Bool_T
```

Type-checking for a simple imperative language

```
env.put ( x, Int_T )
println ( env.get ( x ) // prints Int_T
env.put ( x, Bool_T )
println ( env.get ( x ) // prints Bool_T
```

Alternatively we could throw an exception when adding information about a variable more than once. Various different policies are possible, depending on the details of the language to be typed.

Type-checking for a simple imperative language

```
env.put ( x, Int_T )
println ( env.get ( x ) // prints Int_T
env.put ( x, Bool_T )
println ( env.get ( x ) // prints Bool_T
```

Alternatively we could throw an exception when adding information about a variable more than once. Various different policies are possible, depending on the details of the language to be typed.

If we don't throw an exception, we can define variables more than once in our language. If we do, we have a language where we can only define variables once.

Type-checking for a simple imperative language

We want to write the following method in pseudo-code:

```
Type check ( HashMap<String, Type> env, Prog p ) {
    ...
}
```

It returns the type of `p` under the assumptions (on `p`'s free variables) in `env` if *p* is typeable under these assumptions, otherwise an error should be returned.

Type-checking for a simple imperative language

Translation of

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash 7 : \texttt{Int}}$$

is simple:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p is of form
     BoolLiteral ( b ) then return Bool_T
  else if p is of form
     IntLiteral ( n ) then return Int_T
     ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash 7 : \texttt{Int}}$$

is simple:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p is of form
     BoolLiteral ( b ) then return Bool_T
  else if p is of form
     IntLiteral ( n ) then return Int_T
     ...
}
```

Tricky question: why do we not check `b` and `n`?

Type-checking for a simple imperative language

We want to translate

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P = Q : \texttt{Bool}}$$

to code.

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p is of form
    ...
    Equal ( l, r ) then
        if (check ( env, l ) != Int_T or
            check ( env, r ) != Int_T )
                throw Exception ( "typing error" )
        else return Bool_T
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p of form
    ...
    Ident ( x ) then return env.get ( x )
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \mathtt{Unit} \quad \Gamma \vdash Q : \mathtt{Unit}}{\Gamma \vdash P; Q : \mathtt{Unit}}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p of form
    ...
    Seq ( l, r ) then {
        if ( check ( env, l ) != Unit_T  or
             check ( env, r ) != Unit_T  )
                 throw Exception ( "type error: ..." )
        else
            return Unit_T
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \texttt{Bool} \quad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash \texttt{while}\ P\ \texttt{do}\ \{Q\} : \texttt{Unit}}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p is of form
    ...
    While ( cond, body ) then {
        if ( check ( env, cond ) != Bool_T  or
             check ( env, body ) != Unit_T  )
                 throw Exception ( "type error: ..." )
        else
            return Unit_T
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \texttt{Bool} \quad \Gamma \vdash Q : \alpha \quad \Gamma \vdash R : \alpha \quad \alpha \text{ arbitrary type}}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p of form
    ...
     If ( cond, thenBody, elseBody ) then {
     Type t = check ( env, thenBody )
     if ( check ( env, cond     ) != Bool_T  or
          check ( env, elseBody ) != t  )
                 throw Exception ( "type error: ..." )
     else
            return t
    }
    ...
}
```

Type-checking for a simple imperative language

$$\frac{\Gamma \vdash P : \texttt{Int} \quad \Gamma \vdash Q : \texttt{Int} \quad i \text{ not defined in } \Gamma \quad \Gamma, i : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for } i \;=\; P \texttt{ to } Q \texttt{ do } \{R\} : \texttt{Unit}}$$

translates as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p is of form
    For ( i, from, to, body ) then {
        if ( env.containsKey(i) ) throw Exception(...)
        if ( check ( env, from ) != Int_T or
             check ( env, to   ) != Int_T )
                 throw Exception ( "..." )
        env.put ( i, Int_T  )
        if ( check ( env, body ) != Unit_T  )
             throw Exception ( "..." )
        env.remove ( i )
        else return Unit_T
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \alpha \quad \Gamma \vdash x : \alpha}{\Gamma \vdash x := P : \texttt{Unit}}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog prog ) {
  if prog is of form
    ...
    Assign ( x, p ) then
        if ( check ( env, p ) != env.get ( x ) ) then
            throw Exception ( "..." )
        else
            return Unit_T
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \alpha \quad x \text{ not defined in } \Gamma \quad \Gamma, x : \alpha \vdash Q : \beta}{\Gamma \vdash \mathtt{let}\ \mathtt{x} : \alpha = \mathtt{P}\ \mathtt{in}\ \mathtt{Q} : \beta}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog prog ) {
  if prog of form
    Let ( x, t, p, q ) then {
        if ( env.containsKey(x) ) throw ...
        if ( check ( env, p ) != t ) throw ...
        env.put ( x, t )
        let result = check ( env, q )
        env.remove ( x )
        return result
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if p of form
    ...
    NewList ( t ) then {
        return List_T( t )
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \quad \Gamma \vdash Q : \alpha}{\Gamma \vdash P\texttt{.append(}Q\texttt{)} : \texttt{List<}\alpha\texttt{>}}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog prog ) {
  if prog of form
    ...
    Append ( p, q ) then {
        Type t = check ( env, q )
        if ( check ( env, p ) != List_T( t ) ) throw ...
        return List_T( t )
    }
    ...
}
```

Type-checking for a simple imperative language

Translation of

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \quad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P\texttt{.get(}Q\texttt{)} : \alpha}$$

is as follows:

```
Type check ( HashMap<String, Type> env, Prog prog ) {
  if prog of form
    ...
    Get ( p, q ) then {
        if ( check ( env, q ) != Int_T ) throw ...
        if ( check ( env, p ) =  List_T( t ) )
            return t
        else throw ...

    }
    ...
}
```

A lot more could be said about type-checking

- Typing objects and classes, subtyping (structural vs nominal subtyping)
- Typing methods
- Inheritance
- Traits
- Higher-kinded types
- Types that catch more non-trival bugs, e.g. specifying “this is a sorting function” as types.
- Faster type-checking algorithms
- Type-inference algorithms
- Rust-style lifetime inference
- Gradual typing (Cf Typescript)
- Types for parallel computing
- ...

Conclusion

Types are weak specifications of programs.

Conclusion

Types are weak specifications of programs.

We can check them by walking the AST.

Conclusion

Types are weak specifications of programs.

We can check them by walking the AST.

The key data-structure is the symbol table which holds assumptions about the types of free variables.