The TAL-1 type system

$\tau ::=$	operand types
$\mathsf{Top} \mid \mathsf{Int} \mid \mathsf{Code}(\Gamma)$	
$\mid \mathsf{ptr}(\sigma)$	shared pointer types
$\mid \mathsf{uptr}(\sigma)$	unique pointer types
$\mid \forall \rho \cdot \tau$	quantification over allocated types
$\sigma ::=$	allocated types
ϵ	empty tuple type
τ	one operand
$\langle \sigma_1, \sigma_2 \rangle$	pair
ρ	allocated type variable

operand types are for operands and allocated data types are for tuples.

As before register file types Γ are of the form $\{\mathsf{sp} : \tau, \mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$ where τ, τ_i are operand types.

Similarly heap types Ψ map labels to operand types.

We consider

$$\langle\langle\sigma_1, \sigma_2\rangle, \sigma_3\rangle = \langle\sigma_1, \langle\sigma_2, \sigma_3\rangle\rangle = \langle\sigma_1, \sigma_2, \sigma_3\rangle$$
$$\langle\sigma, \epsilon\rangle = \langle\epsilon, \sigma\rangle = \sigma$$
$$\ldots$$

Typing rules

Typing rules

Tuples

$$\frac{\forall 1 \leq i \leq n \cdot \Psi, \Gamma \vdash \nu_i : \tau_i}{\Psi, \Gamma \vdash \langle \nu_1, \ldots, \nu_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \text{(T-Tuple)}$$

Typing rules

Tuples

$$\frac{\forall 1 \leq i \leq n \cdot \Psi, \Gamma \vdash \nu_i : \tau_i}{\Psi, \Gamma \vdash \langle \nu_1, \ldots, \nu_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \text{(T-Tuple)}$$

$$\frac{\Psi, \Gamma \vdash h : \sigma}{\Psi, \Gamma \vdash \mathsf{uptr}(h) : \mathsf{uptr}(\sigma)} \text{(T-Uptr)}$$

Typing of instructions

The older rules of TAL-0 remain unmodified, except for the Mov instruction, where now copying of unique pointers should be prevented. Hence we have the following new rule.

$$\frac{\Psi, \Gamma \vdash \nu : \tau \quad \tau \neq \mathsf{uptr}(\sigma)}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{(T-Mov1)}$$

Typing of instructions

The older rules of TAL-0 remain unmodified, except for the Mov instruction, where now copying of unique pointers should be prevented. Hence we have the following new rule.

$$\frac{\Psi, \Gamma \vdash \nu : \tau \quad \tau \neq \mathsf{uptr}(\sigma)}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{ (T-Mov1)}$$

We add new typing rules for the new instructions.

$$\frac{\mathsf{n} \geq 0}{\Psi \vdash r_d := \mathsf{malloc}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{r_d : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}} \rangle\}} \text{ (T-Malloc)}$$

malloc creates a unique pointer type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{uptr}(\sigma) \quad r_d \neq \mathsf{sp}}{\Psi \vdash \mathsf{commit}\ r_d : \Gamma \rightarrow \Gamma \oplus \{r_d : \mathsf{ptr}(\sigma)\}} \text{(T-Commit)}$$

commit creates a shared pointer type.

r_d stores a (label) pointer to the value which has now been moved into the heap.

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \to \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-S)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \rightarrow \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-S)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{uptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \rightarrow \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-U)}$$

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau_n \quad \tau_n \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \rightarrow \Gamma} \text{(T-St-S)}$$

Updating shared data should not involve a change in type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau_n \quad \tau_n \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \to \Gamma} \text{(T-St-S)}$$

Updating shared data should not involve a change in type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{uptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau \quad \tau \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \to \Gamma \oplus \{r_d : \mathsf{uptr}\langle \tau_0, \ldots, \tau_{n-1}, \tau, \sigma \rangle\}} \text{(T-St-U)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \qquad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \rightarrow \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\text{ times}}, \sigma \rangle\}} \text{(T-Salloc)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \quad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \rightarrow \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}}, \sigma \rangle\}} \text{(T-Salloc)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}\langle \tau_1, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash \mathsf{sfree}\ \mathsf{n} : \Gamma \rightarrow \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}(\sigma)\}} \text{(T-Sfree)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \qquad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \rightarrow \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}}, \sigma \rangle\}} \text{ (T-Salloc)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}\langle \tau_1, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash \mathsf{sfree}\ \mathsf{n} : \Gamma \rightarrow \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}(\sigma)\}} \text{ (T-Sfree)}$$

Stack underflows are ruled out by the type system.

What about stack overflows??

The type system is not powerful enough to keep track of the size of stack.

Hence Code leading to stack overflow will be well-typed, violating safety.

To ensure type safety, we add new evaluation rules in case of stack overflow.

The type system is not powerful enough to keep track of the size of stack.

Hence Code leading to stack overflow will be well-typed, violating safety.

To ensure type safety, we add new evaluation rules in case of stack overflow.

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle \quad p + n > \mathsf{MaxStack}}{(H, R, \mathsf{salloc}\ n; I) \longrightarrow \mathsf{StackOverflow}} \text{(E-Overflow1)}$$

Where StackOverflow is a new special machine state.

This is similar to "error" terms in our previous discussion on type safety.

The rules for typing instruction sequences, register files, heaps and machine states are as for TAL-0.

We further require rules for quantifying over allocated type variables, and for generating instances.

The rules for typing instruction sequences, register files, heaps and machine states are as for TAL-0.

We further require rules for quantifying over allocated type variables, and for generating instances.

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall \rho \cdot \tau} \text{(T-Gen)}$$

ρ is an allocated type variable possibly occurring in τ.

Type of labels can be instantiated by the following rule.

We replace occurrences of ρ by any desired type τ'.

$$\frac{\Psi, \Gamma \vdash \nu : \forall \rho \cdot \tau}{\Psi, \Gamma \vdash \nu : \tau[\rho \mapsto \tau']} \text{(T-Inst)}$$

Example

$$\begin{aligned} \mathsf{ret0}: \quad & \mathsf{r1} := \mathsf{0}; && \text{// return value} \\ & \mathsf{sfree}\ \mathsf{1}; && \text{// pop argument} \\ & \mathsf{jump}\ \mathsf{r3} && \text{// return} \end{aligned}$$

We would like to assign to this instruction sequence, the type

$\tau = \forall \mathsf{s} \cdot \mathsf{Code}\{\Gamma\}$ where

$\Gamma = \{\mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{s}\rangle, \mathsf{r1}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2}, \mathsf{r3} : \mathsf{Top}\}\}$

where allocated type variable sp represents an arbitrary chunk of memory.

Let $\Gamma_1 = \Gamma \oplus \{\mathsf{r1} : \mathsf{Int}\}$ and $\Gamma_2 = \Gamma_1 \oplus \{\mathsf{sp} : \mathsf{uptr}(\mathsf{s})\}$.

For any heap type Ψ we have the following typing derivation.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2}, \mathsf{r3} : \mathsf{Top}\}\end{array} \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \mathsf{Code}\{\ldots\}}{\Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Sub})}{\Psi \vdash\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Jump})$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Psi,\Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2}, \mathsf{r3} : \mathsf{Top}\}\end{array} \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \mathsf{Code}\{\ldots\}}{\Psi,\Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\text{(T-Sub)}}{\Psi \vdash \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\text{(T-Jump)}$$

$$\dfrac{\dfrac{\Psi,\Gamma_1 \vdash \mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{s}\rangle}{\Psi \vdash \mathsf{sfree}\ 1 : \Gamma_1 \to \Gamma_2}\text{(T-Sfree)} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)\end{array}}{\Psi \vdash \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_1)}\text{(T-Seq)}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := 0 : \Gamma \to \Gamma_1\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_1)\end{array}}{\Psi \vdash \mathsf{r1} := 0; \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma)}\text{(T-Seq)}}{\Psi \vdash \mathsf{r1} := 0; \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \forall \mathsf{s} \cdot \mathsf{Code}(\Gamma)}\text{(T-Gen)}$$

Type Safety for TAL-1

Progress: If $\vdash M$ then there is some M' such that $M \rightarrow M'$.

Preservation: If $\vdash M$ and $M \rightarrow M'$ then either M' is $\mathsf{StackOverflow}$, or $\vdash M'$.

The Java Security Manager

Allows or disallows various operations.

Various kinds of operations (reading or writing files, connecting to another machine) requires asking the security manager for permission.

Security managers are objects of the SecurityManager class.

```
public class BadClass {
  public static void main(String args[]) {
    try {
      Runtime.getRuntime().exec ("/bin/rm /path/to/filexyz");
    } catch (Exception e) {
      System.out.println ("Deletion command failed: " + e);
      return;
    }
    System.out.println ("Deletion command successful!");
  }
}
```

```
public class BadClass {
  public static void main(String args[]) {
    try {
      Runtime.getRuntime().exec ("/bin/rm /path/to/filexyz");
    } catch (Exception e) {
      System.out.println ("Deletion command failed: " + e);
      return;
    }
    System.out.println ("Deletion command successful!");
  }
}
Deletion command successful!
```

The local file gets deleted, if the user has permissions from the operating system.

What if such code is present in some applet loaded by a web-browser?

What if such code is present in some applet loaded by a web-browser?

```
import java.applet.Applet; import java.awt.Graphics;

public class BadApplet extends Applet{

  String text;

  public void init() {
    try { Runtime.getRuntime().exec("/bin/rm -rf /path/to/filexyz");
    } catch (Exception e) { text = "Deletion command failed: " + e; return; }
    text = "Deletion command successful!";
  }

  public void paint(Graphics g){ g.drawString(text, 15, 25); }
}
```

This applet is used in the following HTML page.

```
<html><body>
<applet code="BadApplet.class" width=750 HEIGHT=50></applet>
</body></html>
```

This applet is used in the following HTML page.

```
<html><body>
<applet code="BadApplet.class" width=750 HEIGHT=50></applet>
</body></html>
```

Loading this page in a web browser shows:

```
Deletion command failed: java.security.AccessControlException:
access denied (java.io.FilePermission /bin/rm execute)
```

This applet is used in the following HTML page.

```
<html><body>
<applet code="BadApplet.class" width=750 HEIGHT=50></applet>
</body></html>
```

Loading this page in a web browser shows:

```
Deletion command failed: java.security.AccessControlException:
access denied (java.io.FilePermission /bin/rm execute)
```

The web browser automatically gives restricted permissions to applets.

The sandbox associated with a class depends upon the source from where it was loaded.

The typical sequence used for potentially dangerous operations:

- User program makes some request to the Java API.
- The Java API asks the security manager for permissions.
- If the security manager doesn't want to allow this operation, it throws back an exception which is thrown back to the user program.
- Otherwise the security manager does nothing and the Java API completes the operation.

In the previous example, the user program calls the exec method, which calls the checkExec method on the security manager to check for permission.

The code executed on calling exec is similar to this:

```
public process exec (String command) throws IOException {
  ...
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkExec();
    // security exception can be raised here
  }
  // remaining code follows
  ...
}
```

Another example: reading files.

```
// open a file
FileInputStream fis = new FileInputStream ("somefile");
// read a byte
int x = fis.read();
```

The code executed on calling FileInputStream is similar to

```
public FileInputStream (String name) throws FileNotFoundException {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) { sm.checkRead(name); }
  try { open (name);
  } catch (IOException e) {
    throw new FileNotFoundException (name);
  }
}
```

The System class has various useful data and functions which are global for the whole virtual machine.

The security manager is obtained by getSecurityManager method, and null is returned if no security manager has been set.

The security manager is set by setSecurityManager method, and an exception is raised if the security manager has already been set.

Hence once the security manager has been set, it cannot be modified.

In particular, java applications can set the security manager before executing remote applets, so that these applets don't try to set their own security manager.

Defining one's own security manager: we extend the SecurityManager class and override the functions as required.

```
public class NewSecurityManager extends SecurityManager {
  public void checkExec (String cmd) {
    // always disallow exec
    throw new SecurityException ("exec not allowed")
  }
}
```

Modifying the BadClass to use this security manager.

```
public class NewBadClass {
  public static void main(String args[]) {
    SecurityManager sm = new NewSecurityManager();
    System.setSecurityManager(sm);
    try {
      Runtime.getRuntime().exec ("/bin/rm /path/to/filexyz");
    } catch (Exception e) {
      System.out.println ("Deletion command failed: " + e);
      return;
    }
    System.out.println ("Deletion command successful!");
  }
}
```

Modifying the BadClass to use this security manager.

```
public class NewBadClass {
  public static void main(String args[]) {
    SecurityManager sm = new NewSecurityManager();
    System.setSecurityManager(sm);
    try {
      Runtime.getRuntime().exec ("/bin/rm /path/to/filexyz");
    } catch (Exception e) {
      System.out.println ("Deletion command failed: " + e);
      return;
    }
    System.out.println ("Deletion command successful!");
  }
}
Deletion command failed: java.lang.SecurityException: exec not allowed
```

Examples of methods of the security manager.

- checkRead (String file): called e.g. by FileInputStream (String file).
- checkWrite (String file): called by FileOutputStream (String file).
- checkDelete (String file)

Examples of methods of the security manager.

- checkRead (String file): called e.g. by FileInputStream (String file).
- checkWrite (String file): called by FileOutputStream (String file).
- checkDelete (String file)

Note that while creating a FileInputStream object requires a checkRead call, the actual read() operations on the file input stream requires no permission.

- A trusted class can choose to deliver the FileInputStream object to an untrusted class which can then read from the file.
- It is efficient to check permissions only once.

The Access Controller

- Has functions similar to the security manager.
- Provides easy enforcement of fine grained security policies.
- The security manager works most of the time by calling the access controller.
- Implemented by the AccessController class, accessed through its static methods.

Involves the following four classes.

- The CodeSource class: represents the source from which a certain class was loaded, an an optional list of certificates which was used to sign that code.
- The Permission and Permissions classes: represent various kinds of permissions.
- The Policy class: a policy maps code source objects to permission objects. Only one policy can be associated with the JVM at any point of time, like the security manager. But the policy can be modified.
- The ProtectionDomain class: a protection domain represents all the permissions granted to a particular code source.

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

Permission objects for accessing files are members of the FilePermission class (subclass of the Permission class).

- The type is FilePermission.
- The name is the name of the file.
- Possible actions are "read", "write", "delete" and "execute".

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

Permission objects for accessing files are members of the FilePermission class (subclass of the Permission class).

- The type is FilePermission.
- The name is the name of the file.
- Possible actions are "read", "write", "delete" and "execute".

Permission objects are used for requesting permissions as well as for representing granted permissions.

The security manager, on receiving the checkExec("/bin/rm") call, would normally construct the following permission object

```
FilePermission fp = new FilePermission ("/bin/rm", "execute");
```

and then query the access controller.

```
AccessController.checkPermission (fp);
```

The security manager, on receiving the checkExec("/bin/rm") call, would normally construct the following permission object

```
FilePermission fp = new FilePermission ("/bin/rm", "execute");
```

and then query the access controller.

```
AccessController.checkPermission (fp);
```

Other examples:

```
FilePermission fp1 = new FilePermission ("/bin/*", "execute");
FilePermission fp2 = new FilePermission ("/home/userx", "read, write");
SocketPermission sp1 = new SocketPermission ("hostname:port", "connect");
SocketPermission sp1 = new SocketPermission ("hostname:port", "accept, listen");
```

Policies are specified by objects of Policy class.

It can be obtained and set using getPolicy () and setPolicy (Policy p).

Policy objects can be created by reading from a file which lists the policy rules.

Typically done at startup time:

```
java -Djava.security.manager -Djava.security.policy=<policyfilename> <class> <args>
appletviewer -J-Djava.security.policy=<policyfilename> file.html
```

The policy file have rules mapping code sources to sets of permissions.

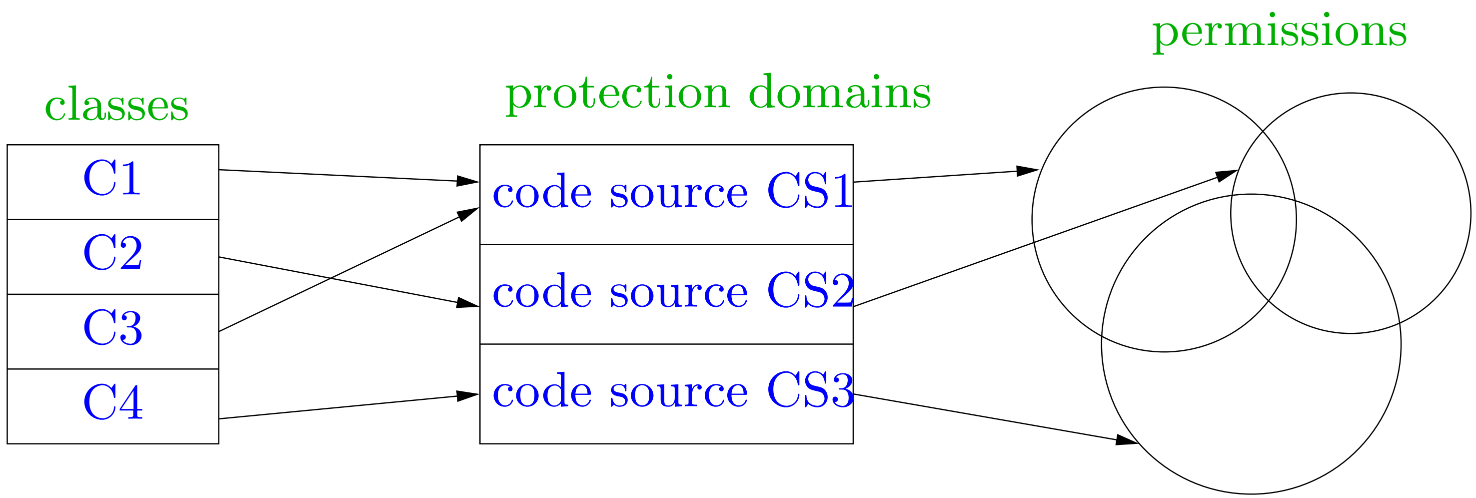
```
grant codeBase "file:/home/userxyz/classes" {
  permission java.io.FilePermission "/bin/rm" "execute";
  permission java.net.SocketPermission "localhost:1024–" "listen, accept";
};

grant signedBy <signer>, codeBase "http://www.xyz.com" {
  permission ...
  ...
};
```

A protection domain groups a code source with a set of permissions.

The class loader is supposed to associate a protection domain with a class when it loads the class.

The protection domain associated with each class is used by the access controller when it is called to check a permission using the checkPermission() method.



Stack inspection

Allowing or disallowing a permission depends on the context in which the checkPermission method was called.

The access controller needs to examine the protection domains associated with all the classes on the stack.

The permission is granted only if all the protection domains on the stack have this permission.

In our old example, the BadClass.main() method for deleting a file calls the Runtime.exec() method which calls the AccessController.checkPermission() to check execute permission on /bin/rm.

Further, the BadClass.main() method itself may be called by some other method m() of class C.

We get the following stack.

AccessController.checkPermission()
Runtime.exec()
BadClass.main()
C.m()
...

The execute permission should be granted only if all the classes on the stack have that permission in their protection domain.

Hence the access controller checks that all frames from the top of the stack to the bottom have this permission in the protection domains of the respective classes.

Sometimes a trusted class may choose to give its permissions to lower frames on the stack.

E.g. an untrusted applet may call some routine to draw something on the screen, and the routine requires some local font file.

This is done using the doPrivileged() method.

```
untrustedclass { f() { ... trustedclass.draw() ...}}
trustedclass {
  public void draw {
    ...
    AccessController.doPrivileged (new PrivilegedAction () {
      public Object run () {
        // privileged code here
        ... <read font file> ...
      } }); }}
```

Instead of the doPrivileged() method

```
AccessController.doPrivileged (new PrivilegedAction () {
  public Object run () {
    <privileged code>
  }
});
```

earlier versions used beginPrivileged() and endPrivileged() calls.

```
AccessController.beginPrivileged();
<privileged code>
AccessController.endPrivileged();
```

To understand the stack inspection algorithm let us assume the following operations.

- enablePrivilege(T)
- disablePrivilege(T)
- checkPrivilege(T)
- revertPrivilege(T)

where T is a target (permission in the Java terminology) we wish to protect.

Actions taken by these operations:

- enablePrivilege(T) puts an enabledPrivilege(T) flag on the current stack frame if the current class has access to T according to the policy.
- disablePrivilege(T) puts a disabledPrivilege(T) flag on the current stack frame (and removes enabledPrivilege(T) flag if present).
- revertPrivilege(T) removes enabledPrivilege(T) and disabledPrivilege(T) flags from the current stack frame if present.
- checkPrivilege(T) examines the stack as follows ...

```
checkPrivilege (T) {
  for SF from top stack frame to bottom stack frame {
    if (policy doesn't allow the class in SF to access T) throw ForbiddenException;
    if (SF has enabledPrivilege (T) flag) return;
    if (SF has disabledPrivilege (T) flag) throw ForbiddedException;
  }
  return; // reached bottom of stack
}
```