2.3 Procedures

We extend our mini-programming language by procedures without parameters and procedure calls.

For that, we introduce a new statement:

\[ f(); \]

Every procedure \( f \) has a definition:

\[ f() \{ \text{stmt}^* \} \]

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure \( \text{main}() \).

In order to optimize such programs, we require an extended operational semantics. :-)

Program executions are no longer paths, but forests:
The function \([\cdot]\) is extended to computation forests: 

\[
[w] : (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})
\]

For a call \(k = (\alpha, f(), v)\) we must:

- determine the initial values for the locals:
  \[
  \text{enter } \rho = \{ x \mapsto 0 \mid x \in \text{Locals} \} \oplus (\rho|_{\text{globals}})
  \]
- ... combine the new values for the globals with the old values for the locals:
  \[
  \text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{locals}}) \oplus (\rho_2|_{\text{globals}})
  \]
- ... evaluate the computation forest inbetween:
  \[
  [k\ (w)] (\rho, \mu) = \begin{cases} 
  \text{let } (\rho_1, \mu_1) = [w] (\text{enter } \rho, \mu) \\
  \text{in } (\text{combine } (\rho, \rho_1), \mu_1)
  \end{cases}
  \]
... in the Example:

The function \([w]\) is extended to computation forests: \(w:\)

\[ [w] : (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z}) \]

For a call \(k = (u, f(), v)\) we must:

- determine the initial values for the locals:
  \(\text{enter } \rho = \{ x \rightarrow 0 \mid x \in \text{Locals} \} \oplus (\rho|_{\text{globals}})\)
- ... combine the new values for the globals with the old values for the locals:
  \(\text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{locals}}) \oplus (\rho_2|_{\text{globals}})\)
- ... evaluate the computation forest inbetween:
  \([k \ (w)] (\rho, \mu) = \text{let } (\rho_1, \mu_1) = [w] (\text{enter } \rho, \mu) \text{ in } \text{combine } (\rho, \rho_1, \mu_1)\)

In order to optimize such programs, we require an extended operational semantics :-(

Program executions are no longer paths, but forests:

... in the Example:
The function \([.]\) is extended to computation forests: \(w:\)

\[ [w] : (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \]

For a call \(k = (u, f(); v)\) we must:

- determine the initial values for the locals:
  
  \[ \text{enter } \rho = \{ x \mapsto 0 \mid x \in \text{Locals} \} \oplus (\rho|_{\text{globals}}) \]

- ... combine the new values for the globals with the old values for the locals:
  
  \[ \text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{locals}}) \oplus (\rho_2|_{\text{globals}}) \]

- ... evaluate the computation forest inbetween:

  \[
  [k \; (w)] \; (\rho, \mu) = \begin{array}{c}
  \text{let } (p_1, \mu_1) = [w] \; (\text{enter } \rho, \mu) \\
  \text{in } \text{combine } (\rho, p_1), p_1
  \end{array}
  \]

---

**Configurations:**

- configuration \(= \text{stack} \times \text{store} \)
- store \(= \text{globals} \times (\mathbb{N} \to \mathbb{Z}) \)
- globals \(= (\text{Globals} \to \mathbb{Z}) \)
- stack \(= \text{frame} \cdot \text{frame}^* \)
- frame \(= \text{point} \times \text{locals} \)
- locals \(= (\text{Locals} \to \mathbb{Z}) \)

A frame specifies the local state of computation inside a procedure call.

The leftmost frame corresponds to the current call.

---

The function \([.]\) is extended to computation forests: \(w:\)

\[ [w] : (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \]

For a call \(k = (u, f(); v)\) we must:

- determine the initial values for the locals:
  
  \[ \text{enter } \rho = \{ x \mapsto 0 \mid x \in \text{Locals} \} \oplus (\rho|_{\text{globals}}) \]

- ... combine the new values for the globals with the old values for the locals:
  
  \[ \text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{locals}}) \oplus (\rho_2|_{\text{globals}}) \]

- ... evaluate the computation forest inbetween:

  \[
  [k \; (w)] \; (\rho, \mu) = \begin{array}{c}
  \text{let } (p_1, \mu_1) = [w] \; (\text{enter } \rho, \mu) \\
  \text{in } \text{combine } (\rho, p_1), p_1
  \end{array}
  \]

---

**Configurations:**

- configuration \(= \text{stack} \times \text{store} \)
- store \(= \text{globals} \times (\mathbb{N} \to \mathbb{Z}) \)
- globals \(= (\text{Globals} \to \mathbb{Z}) \)
- stack \(= \text{frame} \cdot \text{frame}^* \)
- frame \(= \text{point} \times \text{locals} \)
- locals \(= (\text{Locals} \to \mathbb{Z}) \)

A frame specifies the local state of computation inside a procedure call.

The leftmost frame corresponds to the current call.
Computation steps refer to the current call. 

The novel kinds of steps:

\[
\text{call } k = (u, f(); v) : \quad \left( (u, \rho) \cdot \sigma, \gamma, \mu \right) \implies \left( (u_f, \{x \rightarrow 0 \mid x \in \text{Locals}\}) \cdot (v, \rho) \cdot \sigma, \gamma, \mu \right)
\]

\(u_f\) entry point of \(f\)

\[
\text{return: } \quad \left( (r_f, \_), \sigma, \gamma, \mu \right) \implies \left( \sigma, \gamma, \mu \right)
\]

\(r_f\) return point of \(f\)

---

The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

```
1
```

---

The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

```
5 b \rightarrow 0
2
```
The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:
The call stack explicitly implements the DFS traversal through the computation forest: :-)

... in the Example:

```
5  b -> 0
4  b -> 3
3
2
```

1. Idea: Inlining

Copy the procedure body at every call site: !!!

Example:

```c
abs () { 
  a2 = -a1; 
  max () { 
    if (a1 < a2) { ret = a2; goto _exit; } 
    ret = a1; 
  } 
  _exit : 
}
```

... yields:

```c
abs () { 
  a2 = -a1; 
  max () { 
    if (a1 < a2) { ret = a2; goto _exit; } 
    ret = a1; 
  } 
  _exit : 
}
```
Problems:

- The copied block may modify the locals of the calling procedure.
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication.
- How can we handle recursion?

Example:

```c
abs () { 
  a2 = -a1; 
  if (a1 < a2) { ret = a2; goto _exit; }
  ret = a1;
  _exit : 
}
```

Problems:

- The copied block may modify the locals of the calling procedure.
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication.
- How can we handle recursion?
Detection of Recursion:
We construct the call-graph of the program.

In the Examples:

Call-Graph:
- The nodes are the procedures.
- An edge connects $g$ with $h$, whenever the body of $g$ contains a call of $h$.

Strategies for Inlining:
- Just copy nur leaf-procedures, i.e., procedures without further calls :)
- Copy all non-recursive procedures!

... here, we consider just leaf-procedures :)

Transformation 9:

\[ x = 0; \quad (x \in \text{Locals}) \]
Note:

- The Nop-edge can be eliminated if the stop-node of $f$ has no out-going edges ...
- The $x_f$ are the copies of the locals of the procedure $f$.
- According to our semantics of procedure calls, these must be initialized with $0$ :-)

Transformation 9:

2. Idea: Elimination of Tail Recursion

$$f() \{ \text{int } b;$$
$$\quad \text{if } (a_2 \leq 1) \{ \text{ret } a_1; \text{goto } _exit; \}$$
$$\quad b = a_1 \cdot a_2;$$
$$\quad a_2 = a_2 - 1;$$
$$\quad a_1 = b;$$
$$\quad f();$$
$$\text{_exit :}$$
$$\}$$

After the procedure call, nothing in the body remains to be done.

$\Rightarrow$ We may directly jump to the beginning :-)

... after having reset the locals to 0.
... this yields in the Example:

```c
f () { int l;
    _f:
        if (a_2 \leq 1) { ret = a_1; goto _exit; }
        b = a_1 \cdot a_2;
        a_2 = a_2 - 1;
        a_1 = b;
        b = b; goto _f;
    _exit:
}
```

// It works, since we have ruled out references to variables!

---

**Warning:**

- This optimization is crucial for programming languages without iteration constructs !!!
- Duplication of code is not necessary :-)
- No variable renaming is necessary :-)
- The optimization may also be profitable for non-recursive tail calls :-)
- The corresponding code may contain jumps from the body of one procedure into the body of another ????

---

**Transformation 11:**

```
f():
```

---

**Warning:**

- This optimization is crucial for programming languages without iteration constructs !!!
- Duplication of code is not necessary :-)
- No variable renaming is necessary :-)
- The optimization may also be profitable for non-recursive tail calls :-)
- The corresponding code may contain jumps from the body of one procedure into the body of another ????
2. Idea: Elimination of Tail Recursion

\[
\text{f}() \begin{cases}
\text{int } b; \\
\text{if } (a_2 \leq 1) \{ \text{ret } a_1; \text{ goto } _\text{exit}; \} \\
b = a_1 \cdot a_2; \\
a_2 = a_2 - 1; \\
a_1 = b; \\
\text{f}(); \\
\_\text{exit} : \\
\end{cases}
\]

After the procedure call, nothing in the body remains to be done.

\[\Longrightarrow \text{ We may } \text{directly} \text{ jump to the beginning } \sim \text{)}\]

... after having reset the locals to 0.

---

Warning:

\[\rightarrow\text{ This optimization is crucial for programming languages without iteration constructs } \sim\\n\rightarrow\text{ Duplication of code is not necessary } \smiley\\n\rightarrow\text{ No variable renaming is necessary } \smiley\\n\rightarrow\text{ The optimization may also be profitable for non-recursive tail calls } \smiley\\n\rightarrow\text{ The corresponding code may contain jumps from the body of one procedure into the body of another } ???\]

---

... this yields in the Example:

\[
\text{f}() \begin{cases}
\text{int } b; \\
\_f : \text{if } (a_2 \leq 1) \{ \text{ret } a_1; \text{ goto } _\text{exit}; \} \\
b = a_1 \cdot a_2; \\
a_2 = a_2 - 1; \\
a_1 = b; \\
b = 0; \text{ goto } _f; \\
\_\text{exit} : \\
\end{cases}
\]

// It works, since we have ruled out references to variables!

---

Background 4: Interprocedural Analysis

So far, we can analyze each procedure separately.

\[\rightarrow\text{ The costs are moderate } \smiley\]
\[\rightarrow\text{ The methods also work in presence of separate compilation } \sim\]
\[\rightarrow\text{ At procedure calls, we must assume the worst case } \smiley\]
\[\rightarrow\text{ Constant propagation only works for local constants } \smiley\]

Question:

How can recursive programs be analyzed ???
Example: Constant Propagation

```
main() { int t;
    t = 0;
    if (t) x[t] = 3;
    a1 = t; 
    work();
    ret = 1 - ret;
}
```

Example: Constant Propagation

```
main() { 
    if (a1) work(); 
    ret = a1; 
}
```

---

Example: Constant Propagation

```
main() { 
    t = 0;
    if (t) M[x[t]] = 3;
    a1 = t; 
    work();
    ret = 1 - ret;
}
```

Example: Constant Propagation

```
main() { 
    t = 0;
    if (a1) work(); 
    ret = a1; 
}
```

---

(1) **Functional Approach:**

Let \( \mathbb{D} \) denote a complete lattice of (abstract) states.

**Idea:**

Represent the effect of \( f() \) by a function:

\[
[f]^x : \mathbb{D} \rightarrow \mathbb{D}
\]
In order to determine the effect of a call edge \( k = (u, f(\cdot), v) \) we require abstract functions:

\[
\begin{align*}
\text{enter}^k &: \mathcal{D} \rightarrow \mathcal{D} \\
\text{combine}^k &: \mathcal{D}^2 \rightarrow \mathcal{D}
\end{align*}
\]

Then we define:

\[
[k]^2 \mathcal{D} = \text{combine}^k (\mathcal{D}, [f]^2 (\text{enter}^k \mathcal{D}))
\]

... for Constant Propagation:

\[
\begin{align*}
\mathcal{D} &= (\text{Vars} \rightarrow \mathbb{Z}^+) \\
\text{enter}^k \mathcal{D} &= \begin{cases} 
\bot & \text{if } \mathcal{D} = \bot \\
\mathcal{D}|_{\text{Globals}} \oplus \{x \rightarrow 0 \mid x \in \text{Locals}\} & \text{otherwise}
\end{cases} \\
\text{combine}^k (\mathcal{D}_1, \mathcal{D}_2) &= \begin{cases} 
\bot & \text{if } \mathcal{D}_1 = \bot \lor \mathcal{D}_2 = \bot \\
\mathcal{D}_1|_{\text{Locals}} \oplus \mathcal{D}_2|_{\text{Globals}} & \text{otherwise}
\end{cases}
\end{align*}
\]

Example: Constant Propagation

```
main()
  t = 0;

Neg(t)
  M[17] = 3;
  a1 = 1;

Pos(t)
  work();

work();
  ret = 1 - ret;
```

```
work();
  ret = a1;
```

```
work();
  ret = ret;
```
(1) Functional Approach:

Let $D$ denote a complete lattice of (abstract) states.

Idea:

Represent the effect of $f()$ by a function:

$$[f]^D : D \rightarrow D$$

In order to determine the effect of a call edge $k = (n, f(); v)$ we require abstract functions:

$$\text{enter}^D : D \rightarrow D$$

$$\text{combine}^2 : D^3 \rightarrow D$$

Then we define:

$$[k]^D = \text{combine}^2 (D, [f]^D (\text{enter}^D D))$$

... for Constant Propagation:

$$D = (\text{Vars} \rightarrow Z^+)$$

$$\text{enter}^D_D = \begin{cases} \bot & \text{if } D = \bot \\ D|_{\text{Globals}} \oplus \{ \{ x \mapsto 0 \mid x \in \text{Locals} \} \} & \text{otherwise} \end{cases}$$

$$\text{combine}^2 (D_1, D_2) = \begin{cases} \bot & \text{if } D_1 = \bot \lor D_2 = \bot \\ D_1|_{\text{Locals}} \oplus D_2|_{\text{Globals}} & \text{otherwise} \end{cases}$$
In order to determine the effect of a call edge $k = (u, f(), v)$, we require abstract functions:

\[
\text{enter}^4 : \mathbb{D} \to \mathbb{D} \\
\text{combine}^5 : \mathbb{D}^3 \to \mathbb{D}
\]

Then we define:

\[
[k]^2 D = \text{combine}^5 (D, [f]^2 (\text{enter}^4 D))
\]

---

**Observation:**

$\rightarrow$ The effects of assignments are:

\[
[x = e]^2 D = \begin{cases} D \uplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\ D \uplus \{x \mapsto (D y)\} & \text{if } e = y \in \text{Vars} \\ D \uplus \{x \mapsto \top\} & \text{otherwise} \end{cases}
\]

$\rightarrow$ Let \( \mathcal{V} \) denote the (finite !!!) set of constant right-hand sides. Then variables may only take values from \( \mathcal{V} \) 

$\rightarrow$ The occurring effects can be taken from \( D_f \to \mathbb{D}_f \) with \( D_f = (\text{Vars} \to \mathcal{V}) \uplus \)

$\rightarrow$ The complete lattice is huge, but finite !!!

---

**Problems:**

- How can we represent functions \( f : \mathbb{D} \to \mathbb{D} \) ???
- If \( \# \mathbb{D} = \infty \), then \( \mathbb{D} \to \mathbb{D} \) has infinite strictly increasing chains ::(  

**Simplification:** Copy-Constants

$\rightarrow$ Conditions are interpreted as ::)

$\rightarrow$ Only assignments \( x = c \) with \( c \in \text{Vars} \cup \mathbb{Z} \) are treated exactly ::)

---

**Improvement:**

$\rightarrow$ Not all functions from \( D_f \to \mathbb{D}_f \) will occur ::)

$\rightarrow$ All occurring functions \( \lambda D. \perp \neq M \) are of the form:

\[
M = \{ x \mapsto (b \cup \bigcup_{y \in L} y) \mid x \in \text{Vars} \} \\
M D = \{ x \mapsto (b \cup \bigcup_{y \in L} D y) \mid x \in \text{Vars} \} \\
\text{for } D \neq \perp
\]

$\rightarrow$ Let \( \mathcal{M} \) denote the set of all these functions. Then for \( M_1, M_2 \in \mathcal{M} \) \( (M_1 \neq \lambda D. \perp \neq M_2) : 

\[
(M_1 \cup M_2) x = (M_1 x) \cup (M_2 x)
\]

$\rightarrow$ For \( k = \# \text{Vars} \), \( \mathcal{M} \) has height \( O(k^2) \) ::)
Problems:

- How can we represent functions $f : \mathbb{D} \to \mathbb{D}$?
- If $\#\mathbb{D} = \infty$, then $\mathbb{D} \to \mathbb{D}$ has infinite strictly increasing chains.

\[
(\forall x \in \mathbb{Z}) \to (\forall x \in \mathbb{Z})
\]

Simplification: Copy-Constants

\[
\text{Conditions are interpreted as:}
\]

\[
\text{Only assignments } x = c \quad \text{with} \quad c \in \text{Vars} \cup \mathbb{Z} \quad \text{are treated exactly.} \quad \text{(-:)}
\]

562

The effects $[f]^2$ then can be determined by a system of constraints over the complete lattice $\mathbb{D} \to \mathbb{D}$:

\[
\begin{align*}
[v]^F & \equiv \text{id} & v & \text{entry point} \\
[k]^F & \equiv [k]^2 \circ [v]^2 & k = (u, v) & \text{edge} \\
[/]^F & \equiv [\text{stop}]^2 & \text{stop} & \text{end point of } f
\end{align*}
\]

$[v]^F : \mathbb{D} \to \mathbb{D}$ describes the effect of all prefixes of computation forests $w$ of a procedure which lead from the entry point to $f$.

563

Improvement:

- Not all functions from $\mathbb{D}_f \to \mathbb{D}_f$ will occur.
- All occurring functions $\lambda D. \bot \neq M$ are of the form:

\[
M = \{ x \to (b_2 \sqcup \bigcup_{y \in L} y) \mid x \in \text{Vars} \} \quad \text{where:}
\]

\[
M \bot D = \{ x \to (b_2 \sqcup \bigcup_{y \in L} D y) \mid x \in \text{Vars} \} \quad \text{für } D \neq \bot
\]

- Let $\mathcal{M}$ denote the set of all these functions. Then for $M_1, M_2 \in \mathcal{M}$ ($M_1 \neq \lambda D. \bot \neq M_2$):

\[
(M_1 \sqcup M_2) x = (M_1 x) \cup (M_2 x)
\]

- For $k = \#\text{Vars}$, $\mathcal{M}$ has height $O(k^2)$. (-:)}

564

(1) Functional Approach:

Let $\mathbb{D}$ denote a complete lattice of (abstract) states.

Idea:

Represent the effect $f()$ by a function:

$[f]^2 : \mathbb{D} \to \mathbb{D}$

557
Improvement:

→ Not all functions from $D_f \rightarrow D_f$ will occur \(\Rightarrow\)

→ All occurring functions $\lambda D, \bot \not= M$ are of the form:

\[
M = \{ x \mapsto (b_x \cup \bigcup_{y \in L} y) \mid x \in \text{Vars} \}
\]
for $D \not= \bot$

→ Let $M$ denote the set of all these functions. Then for $M_1, M_2 \in M$ $\ (M_1 \not= \lambda D, \bot \not= M_2) :$

\[
(M_1 \cup M_2) x = (M_1 x) \cup (M_2 x)
\]

→ For $k = \#\text{Vars} \ , \ M$ has height $O(k^2) \Rightarrow$

Improvement (Cont.):

→ Also, composition can be directly implemented:

\[
(M_1 \circ M_2) x = b' \cup \bigcup_{y \in L} y \quad \text{with}
\]
\[
b' = b \cup \bigcup_{z \in L} z
\]
\[
I' = \bigcup_{z \in I} I_z
\]

\[
M_1 x = b \cup \bigcup_{z \in I} y
\]
\[
M_2 z = b \cup \bigcup_{z \in L} y
\]

→ The effects of assignments then are:

\[
[x = c]^2 = \begin{cases} 
\text{Id}_\text{Vars} \circ \{ x \mapsto c \} & \text{if } c = c \in \mathbb{Z} \\
\text{Id}_\text{Vars} \circ \{ x \mapsto y \} & \text{if } c = y \in \text{Vars} \\
\text{Id}_\text{Vars} \circ \{ x \mapsto \top \} & \text{otherwise}
\end{cases}
\]