Script generated by TTT

Title: Seidl: Programmoptimierung (09.12.2015)

Date: Wed Dec 09 10:19:18 CET 2015

Duration: 91:56 min

Pages: 54

1.8 Application: Loop-invariant Code

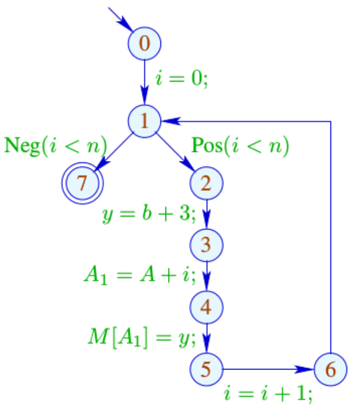
Example

$$\text{for } (i = 0; i < n; i{+}{+})$$
$$a[i] = b + 3;$$

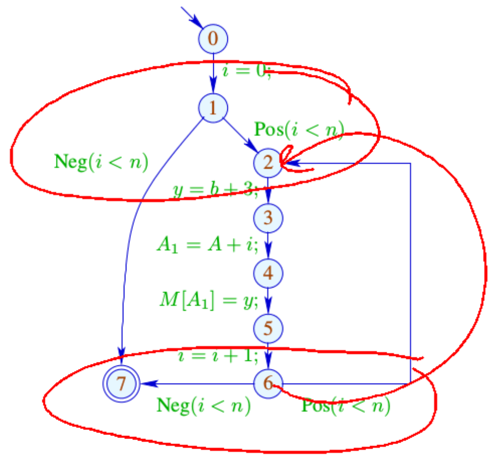
// The expression $b + 3$ is recomputed in every iteration.

// This should be avoided !

The Control-flow Graph



Idea Transform into a do-while-loop ...

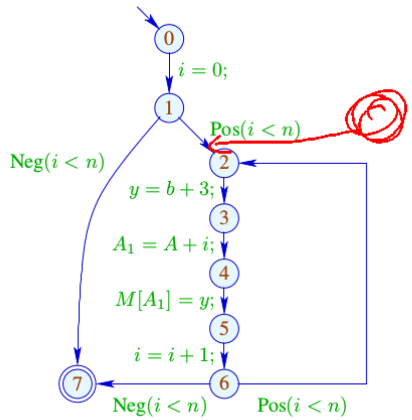


while (e) S

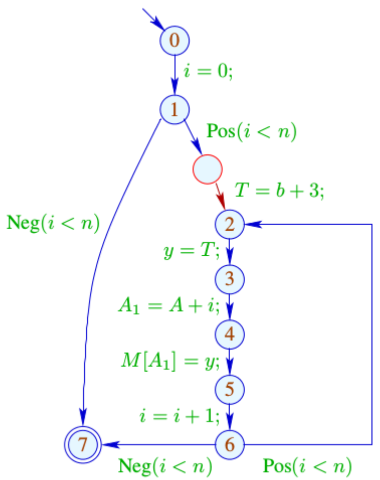
$\Longrightarrow$

if (e) {
do { S while e ;

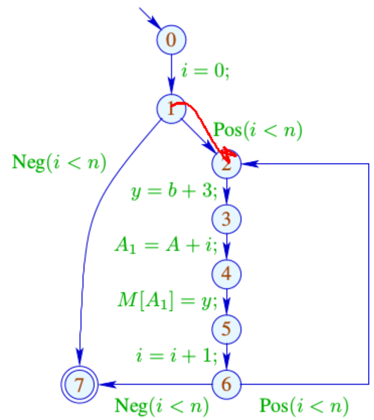
Idea — Transform into a do-while-loop ...



... now there is a place for $T = e;$.

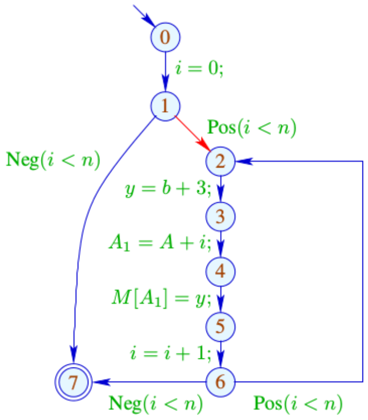


Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
7	$\emptyset$	$\emptyset$

Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
7	$\emptyset$	$\emptyset$

Conclusion

- Elimination of partial redundancies may move loop-invariant code out of the loop.
- This only works properly for do-while-loops !
- To optimize other loops, we transform them into do-while-loops before-hand:

A: while (b) $stmt$ $\Longrightarrow$ if (b)
do $stmt$
while (b);

$\Longrightarrow$ Loop Rotation

Problem

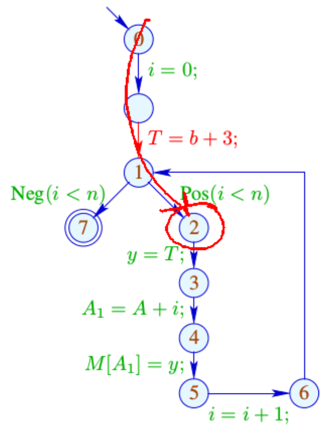
If we do not have the source program at hand, we must re-construct potential loop headers

$\Longrightarrow$ Pre-dominators

u pre-dominates v , if every path $\pi : start \to^* v$ contains u.
We write: $u \Rightarrow v$.

"$\Rightarrow$" is reflexive, transitive and anti-symmetric.

Caveat $T = b + 3;$ may not be placed before the loop :



$\Longrightarrow$ There is no decent place for $T = b + 3;$.

Conclusion

- Elimination of partial redundancies may move loop-invariant code out of the loop.
- This only works properly for do-while-loops !
- To optimize other loops, we transform them into do-while-loops before-hand:

while (b) $stmt$ $\Longrightarrow$ if (b)
do $stmt$
while (b);

$\Longrightarrow$ Loop Rotation

Problem

If we do not have the source program at hand, we must re-construct potential loop headers

$\Longrightarrow$ Pre-dominators

u pre-dominates v , if every path $\pi : start \to^* v$ contains u.
We write: $u \Rightarrow v$.

"$\Rightarrow$" is reflexive, transitive and anti-symmetric.

Computation

We collect the nodes along paths by means of the analysis:

$$\mathbb{P} = 2^{Nodes} \quad , \qquad \sqsubseteq \; = \; \supseteq$$

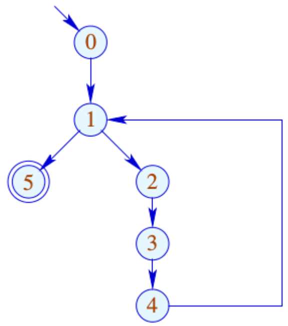
$$[\![(\_, \_, v)]\!]^\sharp \, P \;=\; \boxed{P \cup \{v\}}$$

Then the set $\mathcal{P}[v]$ of pre-dominators is given by:

$$\mathcal{P}[v] = \bigcap \{[\![\pi]\!]^\sharp \, \{start\} \mid \pi : start \to^* v\}$$

Since $[\![k]\!]^\sharp$ are distributive, the $\mathcal{P}[v]$ can computed by means of fixpoint iteration ...

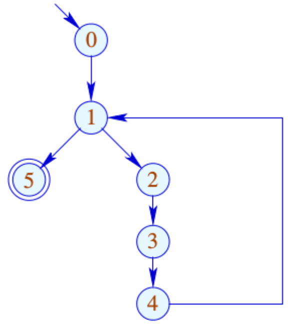
Example



	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

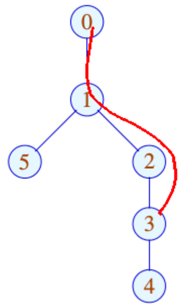
Since $[\![k]\!]^\sharp$ are distributive, the $\mathcal{P}[v]$ can computed by means of fixpoint iteration ...

Example



	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

The partial ordering “$\Rightarrow$” in the example:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0, 1\}$
2	$\{0, 1, 2\}$
3	$\{0, 1, 2, 3\}$
4	$\{0, 1, 2, 3, 4\}$
5	$\{0, 1, 5\}$

Apparently, the result is a tree.

In fact, we have:

Theorem

Every node v has at most one immediate pre-dominator.

Proof

Assume:

there are $u_1 \neq u_2$ which immediately pre-dominate v.

If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable.

Apparently, the result is a tree.

In fact, we have:

Theorem

Every node v has at most one immediate pre-dominator.

Proof

Assume:

there are $u_1 \neq u_2$ which immediately pre-dominate v.

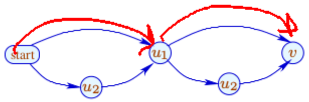
If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable.

Now for every $\pi : start \rightarrow^* v$:

$$\pi = \pi_1 \, \pi_2 \quad \text{with} \quad \begin{array}{l} \pi_1 : start \rightarrow^* u_1 \\ \pi_2 : u_1 \rightarrow^* v \end{array}$$

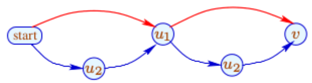
If, however, u_1, u_2 are incomparable, then there is path: $start \rightarrow^* v$ avoiding u_2 :



Now for every $\pi : start \rightarrow^* v$:

$$\pi = \pi_1 \, \pi_2 \quad \text{with} \quad \begin{array}{l} \pi_1 : start \rightarrow^* u_1 \\ \pi_2 : u_1 \rightarrow^* v \end{array}$$

If, however, u_1, u_2 are incomparable, then there is path: $start \rightarrow^* v$ avoiding u_2 :



Observation

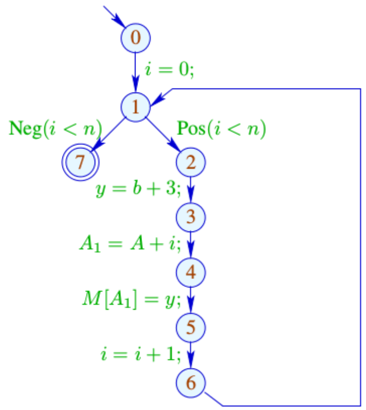
The loop head of a while-loop pre-dominates every node in the body.

A back edge from the exit u to the loop head v can be identified through

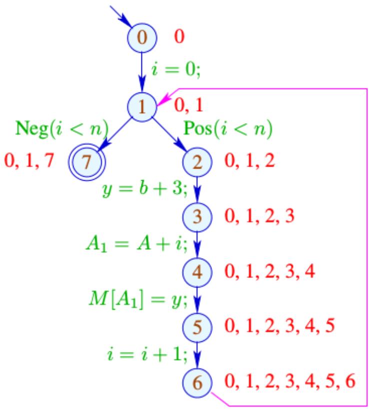
$$v \in \mathcal{P}[u]$$

Accordingly, we define: $(u, \_, v)$

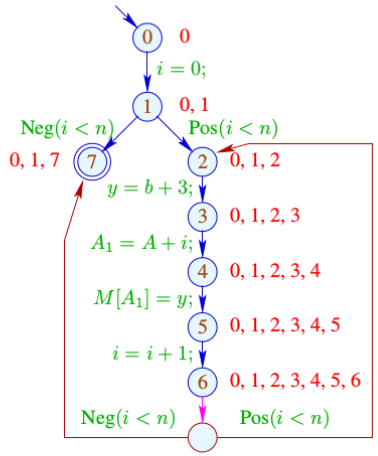
... in the Example



... in the Example

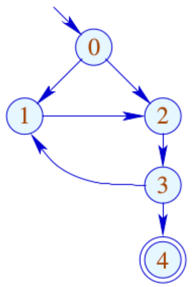


... in the Example

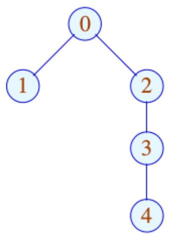


Caveat

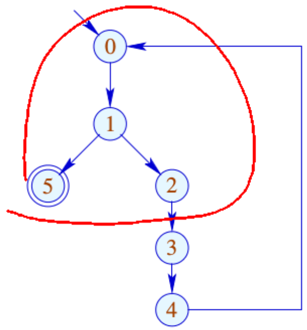
There are unusual loops which cannot be rotated:

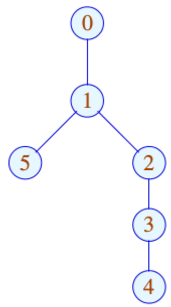


Pre-dominators:



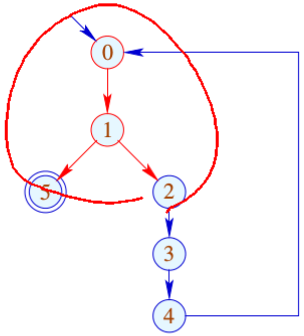
... but also common ones which cannot be rotated:

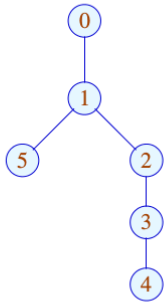




Here, the complete block between back edge and conditional jump should be duplicated.

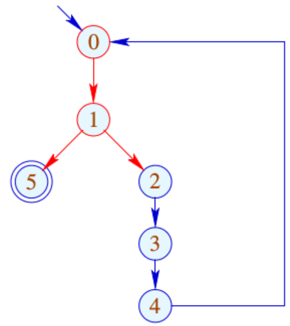
... but also common ones which cannot be rotated:

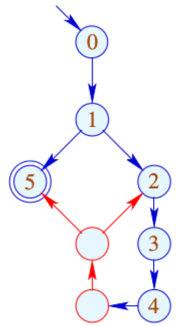




Here, the complete block between back edge and conditional jump should be duplicated.

... but also common ones which cannot be rotated:

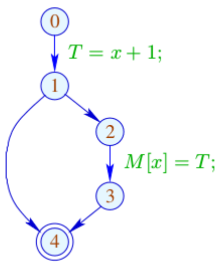




Here, the complete block between back edge and conditional jump should be duplicated.

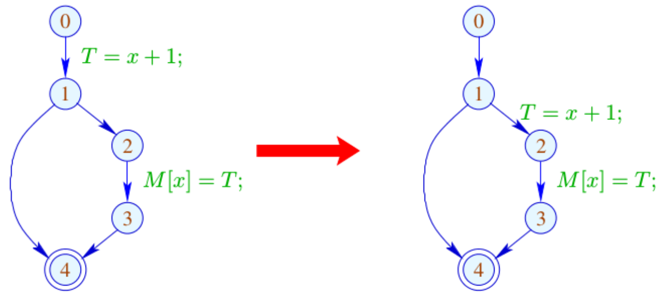
1.9 Eliminating Partially Dead Code

Example



$x + 1$ need only be computed along one path.

Idea



Problem

- The definition $x = e;\quad (x \notin Vars_e)$ may only be moved to an edge where e is safe.
- The definition must still be available for uses of x.

$$\Longrightarrow$$

We define an analysis which maximally delays computations:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & \begin{cases} D\backslash(Use_e \cup Def_x) \cup \{x = e;\} & \text{if} \quad x \notin Vars_e \\ D\backslash(Use_e \cup Def_x) & \text{if} \quad x \in Vars_e \end{cases}
\end{array}
$$

... where:

$$
\begin{array}{lll}
Use_e & = & \{y = e'; \mid y \in Vars_e\} \\
Def_x & = & \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{array}
$$

Problem

- The definition $x = e;\quad (x \notin Vars_e)$ may only be moved to an edge where e is safe.
- The definition must still be available for uses of x.

$$\Longrightarrow$$

We define an analysis which maximally delays computations:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & \begin{cases} D\backslash(Use_e \cup Def_x) \cup \{x = e;\} & \text{if} \quad x \notin Vars_e \\ D\backslash(Use_e \cup Def_x) & \text{if} \quad x \in Vars_e \end{cases}
\end{array}
$$

... where:

$$
\begin{array}{lll}
Use_e & = & \{y = e'; \mid y \in Vars_e\} \\
Def_x & = & \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{array}
$$

For the remaining edges, we define:

$$
\begin{array}{lll}
[\![x = M[e];]\!]^\sharp\, D & = & D\backslash(Use_e \cup Def_x) \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D\backslash(Use_{e_1} \cup Use_{e_2}) \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, D & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, D \quad = \quad D\backslash Use_e
\end{array}
$$

Problem

- The definition $x = e;$ $(x \notin Vars_e)$ may only be moved to an edge where e is safe.
- The definition must still be available for uses of x.

$\Longrightarrow$ $\mathbb{D} = 2^{Ass}, \sqsubseteq = \supseteq$

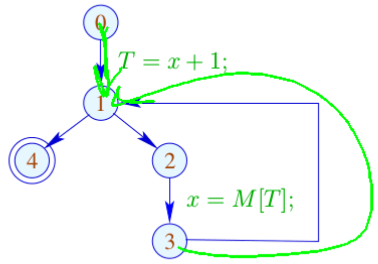
We define an analysis which maximally delays computations:

$$[\![;]\!]^\sharp \, D = D$$

$$[\![x = e;]\!]^\sharp \, D = \begin{cases} D \backslash (Use_e \cup Def_x) \cup \{x = e;\} & \text{if } x \notin Vars_e \\ D \backslash (Use_e \cup Def_x) & \text{if } x \in Vars_e \end{cases}$$

Caveat

We may move $y = e;$ beyond a join only if $y = e;$ can be delayed along all joining edges:



Here, $T = x + 1;$ cannot be moved beyond 1 !!!

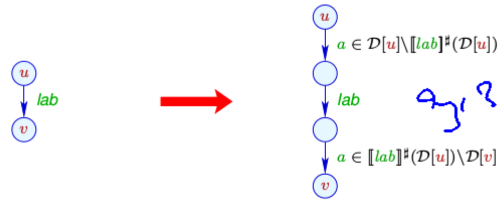
We conclude:

- The partial ordering of the lattice for delayability is given by "$\supseteq$".
- At program start: $D_0 = \emptyset$.

  Therefore, the sets $\mathcal{D}[u]$ of at u delayable assignments can be computed by solving a system of constraints.
- We delay only assignments a where $a\,a$ has the same effect as a alone.
- The extra insertions render the original assignments as assignments to dead variables ...

Transformation 7

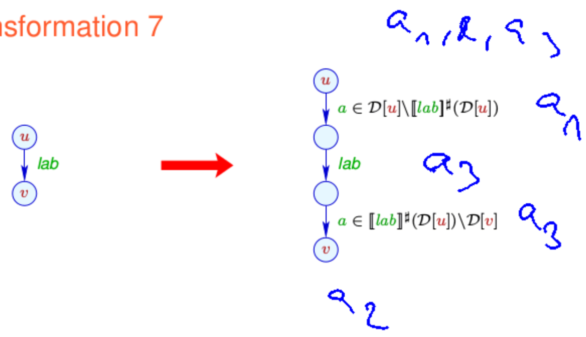
$\mathcal{D}[u] = \{a_1, a_2\}$

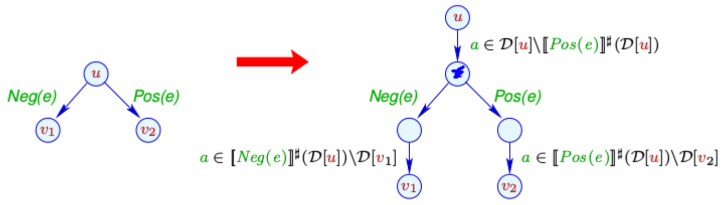


$a_1 ?$

$\mathcal{D}(\bar{v}) = \{a_2\}$

Transformation 7

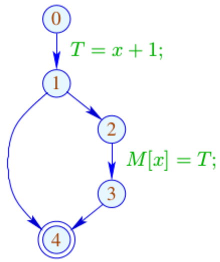




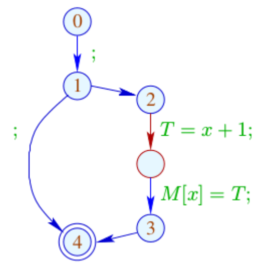
Remark

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2.

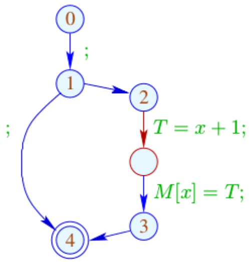
In the example, the partially dead code is eliminated:



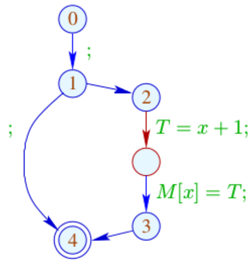
	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$

474

Remarks

- After $T7$, all original assignments $y = e;$ with $y \notin Vars_e$ are assignments to dead variables and thus can always be eliminated.
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code.
- Similar to the elimination of partial redundancies, the transformation can be repeated.

Conclusion

→ The design of a meaningful optimization is non-trivial.

→ Many transformations are advantageous only in connection with other optimizations !

→ The ordering of applied optimizations matters !!

→ Some optimizations can be iterated !!!

... a meaningful ordering:

T4	Constant Propagation Interval Analysis Alias Analysis
T6	Loop Rotation
T1, T3, T2	Available Expressions
T2	Dead Variables
T7, T2	Partially Dead Code
T5, T3, T2	Partially Redundant Code

... a meaningful ordering:

T4	Constant Propagation Interval Analysis Alias Analysis
T6	Loop Rotation
T1, T3, T2	Available Expressions
T2	Dead Variables
T7, T2	Partially Dead Code
T5, T3, T2	Partially Redundant Code

2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Evaluation of Polynomials

$$f(x) = \boxed{a_n} \cdot \boxed{x^n} + \boxed{a_{n-1} \cdot x^{n-1}} + \ldots \boxed{+ a_1 \cdot x} + a_0$$

	Multiplications	Additions
naive	$\boxed{\frac{1}{2}n(n+1)}$	$\boxed{n}$
re-use	$\boxed{2n-1}$	$\boxed{n}$
Horner-Scheme	n	n

Idea

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabulation of a polynomial $f(x)$ of degree n :

→ To recompute $f(x)$ for every argument x is too expensive.

→ Luckily, the n-th differences are constant !!!

2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Evaluation of Polynomials

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

	Multiplications	Additions
naive	$\frac{1}{2}n(n+1)$	n
re-use	$2n-1$	n
Horner-Scheme	n	n

Idea

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabulation of a polynomial $f(x)$ of degree n :

→ To recompute $f(x)$ for every argument x is too expensive.

→ Luckily, the n-th differences are constant !!!

Example: $f(x) = 3x^3 - 5x^2 + 4x + 13$

n	$f(n)$	Δ	Δ^2	Δ^3
0	13	2	8	$\boxed{18}$
1	15	10	$\boxed{26}$	
2	25	$\boxed{36}$	44	
3	$\boxed{61}$	86		
4	141			

Here, the n-th difference is always

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ step width})$$