Warning:

- Our representation of numbers is not unique: \(011101\) should be accepted iff every word from \(011101\cdot 0^*\) is accepted!
- This property is preserved by union, intersection and complement :-(
- It is lost by projection !!!

\[ \longrightarrow \text{The automaton for projection must be enriched such that the property is re-established}!! \]

Observation:

The set of satisfying variable assignments is regular :))

\[
\begin{align*}
\phi_1 \land \phi_2 & \rightarrow L(\phi_1) \cap L(\phi_2) \quad \text{(Intersection)} \\
\neg \phi & \rightarrow L(\phi) \quad \text{(Complement)} \\
\exists x : \phi & \rightarrow L(\phi)(x) \quad \text{(Projection)}
\end{align*}
\]

\[ \begin{align*}
\text{\( x \gg 20 \)} \\
\text{\( x = 20 + s \)}
\end{align*} \]
Projecting away the $x$-component:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>213</td>
<td>$t$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>$z$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>89</td>
<td>$y$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>$x$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Warning:

- Our representation of numbers is not unique: 011101 should be accepted iff every word from 011101 · 0$^*$ is accepted!
- This property is preserved by union, intersection and complement :-(
- It is lost by projection !!!

⇒ The automaton for projection must be enriched such that the property is re-established!!

Automata for Basic Predicates:

$x = 5$
Automata for Basic Predicates:

\[ x + x = y \]

Results:

Ferrante, Rackoff, 1973: \( \text{PSAT} \leq \text{DSPACE}(2^{2^n}) \)

Fischer, Rabin, 1974: \( \text{PSAT} \geq \text{NTIME}(2^{2^n}) \)
Results:

Ferrante, Rackoff, 1973 : \( \text{PSAT} \leq \text{DSpace}(2^{2^n}) \)

Fischer, Rabin, 1974 : \( \text{PSAT} \geq \text{NTIME}(2^{2^n}) \)

3.3 Improving the Memory Layout

Goal:
- Better utilization of caches
  \( \Rightarrow \) reduction of the number of cache misses
- Reduction of allocation/de-allocation costs
  \( \Rightarrow \) replacing heap allocation by stack allocation
  \( \Rightarrow \) support to free superfluous heap objects
- Reduction of access costs
  \( \Rightarrow \) short-circuiting indirection chains (Unboxing)

Possible Solutions:

\( \Rightarrow \) Reorganize the data accesses!
\( \Rightarrow \) Reorganize the data!

Such optimizations can be made fully automatic only for arrays \( \text{arrays} \! : \! (\text{arrays}) \).

Example:

\[
\begin{align*}
\text{for } & (j = 1; j < n; j++) \\
\text{for } & (i = 1; i < m; i++) \\
& a[i][j] = a[i - 1][j - 1] + a[i][j];
\end{align*}
\]
At first, always iterate over the rows!

Exchange the ordering of the iterations:

\[
\begin{align*}
&\text{for } (i = 1; i < m; i++) \\
&\text{for } (j = 1; j < n; j++) \\
&\quad a[i][j] = a[i - 1][j - 1] + a[i][j];
\end{align*}
\]

When is this permitted???
In our case, we must check that the following equation systems have no solution:

\[
\begin{array}{c|c}
\text{Write} & \text{Read} \\
\hline
(i_1, j_1) & (i_2 - 1, j_2 - 1) \\
\vdots & \vdots \\
(i_2, j_1) & (i_2 - 1, j_2 - 1) \\
\end{array}
\]

The first implies: \( j_2 \leq j_3 - 1 \) Hurra!
The second implies: \( i_2 \leq i_3 - 1 \) Hurra!

---

Example: 

Matrix-Matrix Multiplication

\[
\begin{align*}
\text{for } & (i = 0; i < N; i++) \\
\text{for } & (j = 0; j < M; j++) \\
\text{for } & (k = 0; k < K; k++) \\
& c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\end{align*}
\]

Over \( b[][] \) the iteration is columnwise :-(

---

Exchange the two inner loops:

\[
\begin{align*}
\text{for } & (i = 0; i < N; i++) \\
\text{for } & (k = 0; k < K; k++) \\
\text{for } & (j = 0; j < M; j++) \\
& c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\end{align*}
\]

Is this permitted ???
Exchange the two inner loops:

for \( i = 0; i < N; i++ \)
for \( k = 0; k < K; k++ \)
for \( j = 0; j < M; j++ \)

\[ c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \]

Is this permitted ???

Discussion:

- Correctness follows as before  :-) 
- A similar idea can also be used for the implementation of multiplication for row compressed matrices  :-))
- Sometimes, the program must be massaged such that the transformation becomes applicable  :-(
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...
Exchange the two inner loops:

```c
for (i = 0; i < N; i++)
    for (k = 0; k < K; k++)
        for (j = 0; j < M; j++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Is this permitted ???

```c
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        c[i][j] = 0;
    for (k = 0; k < K; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

- Now, the two iterations can no longer be exchanged :-(
- The iteration over `j`, however, can be duplicated ...

Discussion:

- Correctness follows as before :-)
- A similar idea can also be used for the implementation of multiplication for row compressed matrices :-))
- Sometimes, the program must be massaged such that the transformation becomes applicable :-(
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...

```c
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        c[i][j] = 0;
    for (j = 0; j < M; j++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Correctness:

- The read entries (here: no) may not be modified in the remaining body of the loop !!!!
- The ordering of the write accesses to a memory cell may not be changed :-)
for $i = 0; i < N; i++$
  for $j = 0; j < M; j++$
    $c[i][j] = 0$;
    for $k = 0; k < K; k++$
      $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]$;
  }

- Now, the two iterations can no longer be exchanged :-(
- The iteration over $j$, however, can be duplicated ...

We obtain:

for $i = 0; i < N; i++$
  for $j = 0; j < M; j++$
    $c[i][j] = 0$;
    for $k = 0; k < K; k++$
      for $j = 0; j < M; j++$
        $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]$;
  }

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-)
- Accordingly, conditionals may be moved out of the loop if-distribution ...

We obtain:

for $i = 0; i < N; i++$
  for $j = 0; j < M; j++$
    $c[i][j] = 0$;
    for $k = 0; k < K; k++$
      for $j = 0; j < M; j++$
        $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]$;
  }

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-)
- Accordingly, conditionals may be moved out of the loop if-distribution ...
Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

\[
\begin{align*}
\text{for} \ (i = 0; i < N; i++) & \\
\quad \text{for} \ (j = 0; j < M; j++) & \\
\qquad t = 0; \\
\qquad \text{for} \ (k = 0; k < K; k++) & \\
\qquad\qquad t &= t + a[i][k] \cdot b[k][j]; \\
\qquad c[i][j] &= t; \\
\end{align*}
\]

Idea:

If we find heavily used array elements \( a[i_1] \ldots [i_e] \) whose index expressions stay constant within the inner loop, we could instead provide auxiliary registers :-)

Warning:

The latter optimization prohibits the former and vice versa ...

Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

\[
\begin{align*}
\text{for} \ (i = 0; i < N; i++) & \\
\quad \text{for} \ (j = 0; j < M; j++) & \\
\qquad t = 0; \\
\qquad \text{for} \ (k = 0; k < K; k++) & \\
\qquad\qquad t &= t + a[i][k] \cdot b[k][j]; \\
\qquad c[i][j] &= t; \\
\end{align*}
\]

Correctness:

\[\rightarrow\] The read entries (here: no) may not be modified in the remaining body of the loop !!!!

\[\rightarrow\] The ordering of the write accesses to a memory cell may not be changed :-)
Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

\[
\text{for } (i = 0; i < N; i++) \\
\text{for } (j = 0; j < M; j++) \\
\quad t = 0; \\
\quad \text{for } (k = 0; k < K; k++) \\
\quad \quad t = t + a[i][k] * b[k][j]; \\
\quad c[i][j] = t;
\]

Idea:

If we find heavily used array elements \(a[e_1] \ldots [e_s]\) whose index expressions stay constant within the inner loop, we could instead also provide auxiliary registers :-)

Warning:

The latter optimization prohibits the former and vice versa ...

Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic ...

Example: Stacks

\[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \bullet\]

Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic ...

Example: Stacks

\[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \bullet\]
Alternative:

```
  s  sp  a
  1  2  3  4
```

**Advantage:**

+ The implementation is also simple  :-)
+ The operations push / pop still require constant time  :-)
+ The data are consecutively allocated; stack oscillations are typically small
  
  ➞ better Cache behavior !!!

**Discussion:**

- The same idea also works for queues  :-)
- Other data-structures are attempted to organize blockwise.
  
  **Problem:** how can accesses be organized such that they refer mostly to the same block ???

  ➞ Algorithms for external data

**2. Stack Allocation instead of Heap Allocation**

**Problem:**

- Programming languages such as Java allocate all data-structures in the heap — even if they are only used within the current method  :(.
- If no reference to these data survives the call, we want to allocate these on the stack  :-)

  ➞ Escape Analysis

**Idea:**

Determine points-to information.
Determine if a created object is possibly reachable from the outside...

**Example:**

Our Pointer Language

```
x = new();
y = new();
x[A] = y;
z = y;
ret;
```

... could be a possible method body  :-)
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`; or
- are reachable from global variables.

... in the Example:

```plaintext
x = new();
y = new();
x[A] = y;
z = y;
ret = [ ];
```

We conclude:

- The objects which have been allocated by the first `new()` may never escape.
- They can be allocated on the stack  :-)

Warning:

This is only meaningful if only few such objects are allocated during a method call  :-(

If a local `new()` occurs within a loop, we still may allocate the objects in the heap  :-(

Extension: Procedures

- We require an interprocedural points-to analysis  :-)  
- We know the whole program, we can, e.g., merge the control-flow graphs of all procedures into one and compute the points-to information for this.
- Warning: If we always use the same global variables `y_1, y_2, ...` for the simulation of parameter passing, the computed information is necessarily imprecise  :-(
- If the whole program is not known, we must assume that each reference which is known to a procedure escapes  :-(