Algorithmic Species Revisited: A Program Code Classification Based on Array References

Cedric Nugteren (presenter), Rosilde Corvino, Henk Corporaal

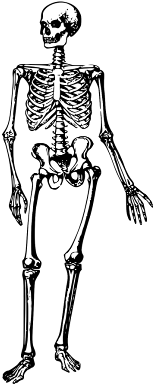
Eindhoven University of Technology (TU/e)
http://parse.ele.tue.nl/
c.nugteren@tue.nl

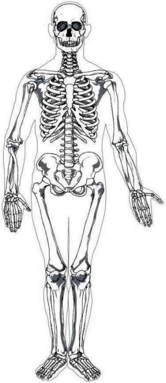
September 7, 2013

Species and skeletons

Are these two actors of the same species?

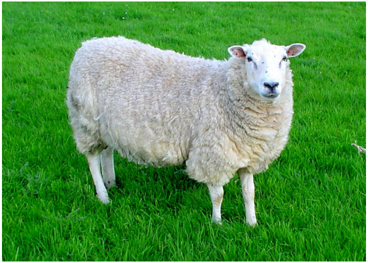
Species and skeletons





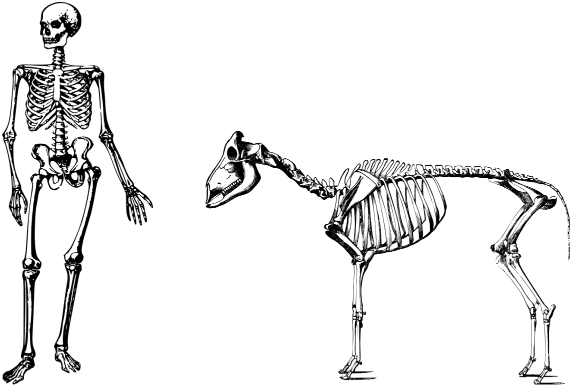
They are. Possible explanation: their skeletons look alike.

Species and skeletons

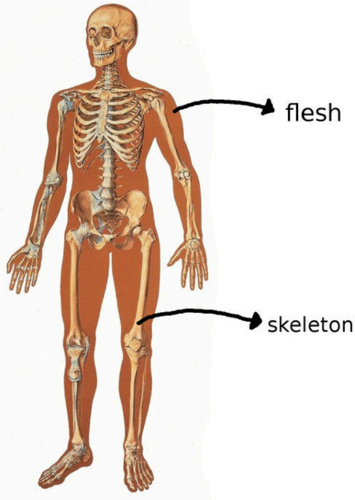
And what about these two?

Species and skeletons



They are not: their skeleton is quite different.

Species and skeletons



Functionality:
what you want to compute
e.g. the sum of a vector

Structure:
parallelism, memory access patterns
e.g. parallel reduction tree, data reuse

Algorithmic species

Algorithmic species:

- Classification based on memory access patterns and parallelism
- Is formally defined based on the polyhedral model
- Can be extracted automatically or used manually
- To be used:
  1. In skeleton-based compilers (automatic)
  2. For performance prediction (automatic/manual)
  3. As design patterns (manual)

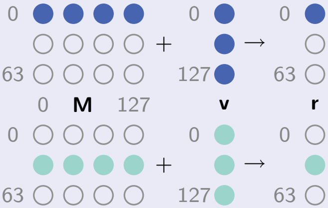
For more information on species and skeletons:

1. C. Nugteren, P. Custers, and H. Corporaal. **Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming**. In *ACM TACO*. 2013.
2. C. Nugteren, P. Custers, and H. Corporaal. **Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification**. In *APPT*. Springer, 2013.

Example algorithmic species

Matrix-vector multiplication:

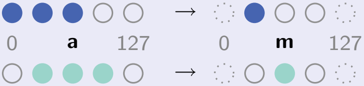
```
for (i=0; i<64; i++) {
  r[i] = 0;
  for (j=0; j<128; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```



M[0:63,0:127]|chunk(-,0:127) ∧ v[0:127]|full → r[0:63]|element

Stencil computation:

```
for (i=1; i<128-1; i++) {
  m[i] = 0.33 * (a[i-1]+a[i]+a[i+1]);
}
```



a[1:126]|neighbourhood(-1:1) → m[1:126]|element

Motivation

1a. Can't we unify the patterns?

- *Element* is a special case of *neighbourhood* or *chunk*
  A[0:N,0:M]|element = A[0:N,0:M]|chunk(-,-) = A[0:N,0:M]|neighb(0:0,0:0)
- We cannot represent a *chunk* pattern with overlap:
  we would need a neighbourhood-chunk combination

1b. Can't we apply the theory for non static affine loop nests?

- The species-theory is limited to code that fits the polyhedral model
- Automatic extraction will not always be possible...
  ... at least manual classification should be!

2. Can't we capture more details?

- Some pairs of code have significantly different access patterns (and performance), but belong to the same species
- Example: loop tiling (discussed later on)

Outline

Outline

Species revisited

Overview of the new theory

- Characterise individual array references
- Merge characterisations
- Translate characterisations into species

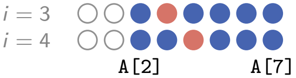
(automated through A-DARWIN)

Array reference characterisation

$\mathcal{R} = (\mathcal{N}, \mathcal{A}, \mathcal{D}^N, \mathcal{E}^N, \mathcal{S}^N) \quad \rightarrow \quad$ (name, r/w, domain, size, step)

First example

```
for (i=2; i<8; i++)
  B[i-2] = A[i];
```



Array reference characterisation

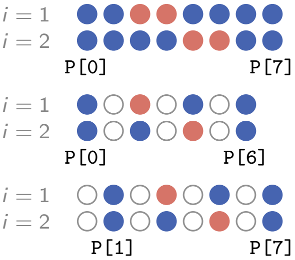
`A[i]` $(A, r, [2..7], 1, 1)$

`B[i-2]` $(B, w, [0..5], 1, 1)$

Second example

```
for (i=0; i<4; i++)
  Q[i] = 0;
  for (j=0; j<2; j++)
    Q[i] += P[2*i+j];
```

```
for (i=0; i<4; i++)
  Q[i] = P[2*i] + P[2*i+1];
```



Array reference characterisation (for P only)

First loop:

P[2*i+j] $(P, r, [0..7], 2, 2)$

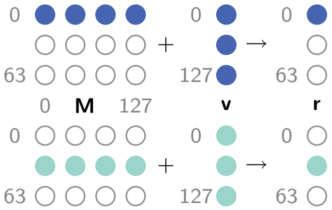
Second loop:

P[2*i] $(P, r, [0..6], 1, 2)$

P[2*i+1] $(P, r, [1..7], 1, 2)$

Matrix-vector multiplication

```
for (i=0; i<64; i++) {
  r[i] = 0;
  for (j=0; j<128; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```



Array reference characterisation

M[i][j] $(M, r, \langle[0..63][0..127]\rangle, \langle 1, 128\rangle, \langle 1, 0\rangle) \longrightarrow$ M[0:63,0:127]chunk(−,0:127)

v[j] $(v, r, [0..127], 128, 0) \longrightarrow$ v[0:127]full

r[i] $(r, w, [0..63], 1, 1) \longrightarrow$ r[0:63]element

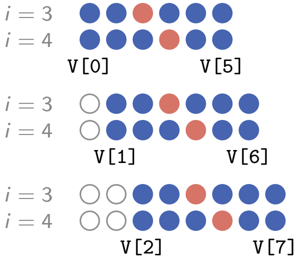
Merging algorithm

Input: array references R (w.r.t. a loop nest)

```
foreach {R_a, R_b} ∈ R do
    if N_a = N_b and A_a = A_b and S_a = S_b then
        if |D_a| = |D_b| and D_a ∩ D_b ≠ ∅ then
            D_new = D_a ∪ D_b
            E_new = |min(D_a) − min(D_b)|
            if E_a + E_b + t_gap > E_new then
                R_new = (N_a, A_a, D_new, E_new, S_a)
                replace R_a and R_b with R_new in R
            end
        end
    end
end
```

Merging example

```
for (i=1; i<7; i++) {
  W[i] = V[i-1] +
         V[i] +
         V[i+1];
}
```



Array reference characterisation

Before merging:

`V[i-1]` $(V, r, [0..5], 1, 1)$

`V[i]` $(V, r, [1..6], 1, 1)$

`V[i+1]` $(V, r, [2..7], 1, 1)$

After merging:

`V[]` $(V, r, [0..7], 3, 1)$

Translating into species

Input: array references R after merging (w.r.t. a loop nest)
$X = \emptyset$
foreach $\mathcal{R}_a \in R$ **do**
    if $\mathcal{S}_a = 0$ **and** $\mathcal{A}_a = r$ **then**
        $X \leftarrow \mathcal{N}_a\ \mathcal{D}_a$ full
    else if $\mathcal{S}_a = 0$ **and** $\mathcal{A}_a = w$ **then**
        $X \leftarrow \mathcal{N}_a\ \mathcal{D}_a$ shared
    else if $\mathcal{E}_a = 1$ **then**
        $X \leftarrow \mathcal{N}_a\ \mathcal{D}_a$ element
    else if $\mathcal{S}_a < \mathcal{E}_a$ **then**
        $X \leftarrow \mathcal{N}_a\ \mathcal{D}_a$ neighbourhood $(\mathcal{E}_a)$
    else
        $X \leftarrow \mathcal{N}_a\ \mathcal{D}_a$ chunk $(\mathcal{E}_a)$
    end
end

Information is lost in the translation at the cost of readability

Beyond static affine loop nests

Beyond static affine loop nests

- The classification is an over-approximation: it gives an upper-bound
- Automatic classification (using A-DARWIN) is not always possible:
  - Either an upper-bound is given or ...
  - ... manual classification can be applied

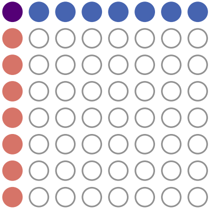
Outline

First example: row-major versus column-major

Array reference characterisation extended → SPECIES+

$$\mathcal{R} = (\mathcal{N}, \mathcal{A}, \mathcal{D}^N, \mathcal{E}^N, \mathcal{S}^N) \quad \rightarrow \quad (\mathcal{N}, \mathcal{A}, \mathcal{D}^N, \mathcal{E}^N, \mathcal{S}^{N,M}, \mathcal{X}^M)$$

```
for (i=0; i<8; i++)
  for (j=0; j<8; j++)
    ... = X[i*8+j] + X[j*8+i];
```



Array reference characterisation

Before:

`X[]` $(X, r, [0..63], 1, 1)$

With finer-grained SPECIES+:

`X[i*8+j]` $(X, r, [0..63], 1, 8|1, 8|8)$

`X[j*8+i]` $(X, r, [0..63], 1, 1|8, 8|8)$

Second example: tiling

```
for (i=0; i<8; i++)
  for (j=0; j<8; j++)
    E[i][j] = 0;
```

```
for (i=0; i<8; i=i+2)
  for (j=0; j<8; j=j+2)
    for (ii=0; ii<2; ii++)
      for (jj=0; jj<2;jj++)
        E[i+ii][j+jj] = 0;
```

Array reference characterisation

Un-tiled (with SPECIES+):

`E[i][j]` $(E, w, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1|0, 0|1\rangle, 8|8)$

Tiled (with SPECIES+):

`E[i+ii][j+jj]` $(E, w, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 2|0|1|0, 0|2|0|1\rangle, 4|4|2|2)$

Outline

Summary

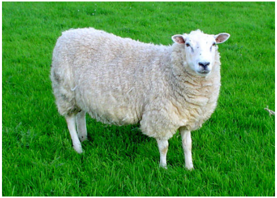
The revised classification 'algorithmic species':

- Captures memory access patterns from C source code
- Uses array reference characterisations as 'unified patterns'
- Can be applied for non static affine loop nests
- Automates classification through A-DARWIN

The extended classification SPECIES+:

- Captures an increased amount of performance-relevant details
- ...but is less readable and intuitive

Questions / further information



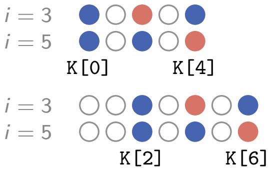
Thank you for your attention!

A-DARWIN is available at:
http://parse.ele.tue.nl/species/

For more information and links to publications, visit:
http://parse.ele.tue.nl/
http://www.cedricnugteren.nl/

Additional merging example: interpolation

```
for (i=1; i<6; i+=2) {
  L[i] = K[i-1] + K[i+1];
}
```



Array reference characterisation

Before merging:

`K[i-1]` $(K, r, [0..4], 1, 2)$

`K[i+1]` $(K, r, [2..6], 1, 2)$

After merging (optional):

`K[]` $(K, r, [0..6], 3, 2)$

Beyond static affine loop nests

```
// Non-static control
while(i<8) {
  B[i] = A[i];
  i = i + A[i];
}
```

```
// Non-affine condition
for(i=0; i<8; i++) {
  if (P[i] > 12)
    P[i] = 0;
}
```

```
// Non-affine bound
for(i=0; i<8-i*i; i++)
  H[0] = G[i];
```

```
// Non-affine references
for(i=0; i<8; i++)
  S[T[i]] = R[i*i];
```

- Non-static control: Not trivially parallelisable
- Non-affine bounds: Upper-bound on domain
  $(G, r, [0..3], 1, 1)$
- Non-affine conditions: Upper-bound on step and domain
  $(P, w, [0..7], 1, 1)$
- Non-affine references: Upper-bound on step and domain
  $(R, r, [0..49], 1, 1)$ and $(S, w, [0..255], 256, 0)$