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

Cedric Nugteren, Rosilde Corvino, and Henk Corporaal
Eindhoven University of Technology, The Netherlands
c.nugteren@tue.nl, r.corvino@tue.nl, h.corporaal@tue.nl

***Abstract*—The shift towards parallel processor architectures has made programming, performance prediction and code generation increasingly challenging. Abstract representations of program code (i.e. classifications) have been introduced to address this challenge. An example is '*algorithmic species*', a memory access pattern classification of loop nests. It provides an architecture-agnostic structured view of program code, allowing programmers and compilers to take for example parallelisation decisions or perform memory hierarchy optimisations.**

The existing algorithmic species theory is based on the polyhedral model and is limited to static affine loop nests. In this work, we first present a revised theory of algorithmic species that overcomes this limitation. The theory consists of a 5-tuple characterisation of individual array references and their corresponding merging operation. Second, we present an extension of this theory named SPECIES+, providing a more detailed 6-tuple characterisation. With this, we are able to retain relevant access pattern information not captured by the original algorithmic species, such as column-major versus row-major matrix accesses. We implement both new theories into a tool, enabling automatic classification of program code.

I. Introduction

Programming, performance prediction, and code generation have become increasingly challenging over the last decade due to the shift towards parallel processor architectures [9]. Examples are multi-core CPUs, graphics processing units (GPUs), and many-core architectures such as the Kalray MPPA and Intel MIC. It has become paramount for programmers, performance models, and compilers to carefully deal with the parallelism and multi-level memory hierarchy exposed by these architectures, especially considering the *memory wall* [9], [18] and the prospect of *dark silicon* [7].

In this work, we revisit *'algorithmic species'* [13], a memory access pattern based classification targeted at parallel architectures such as GPUs. Algorithmic species provide an architecture-agnostic structured classification (or *summary*) of code. Programmers and compilers can use this for example to take parallelisation decisions or to perform memory hierarchy optimisations. However, in this work we only present the classification itself and not the optimisations.

The theory behind algorithmic species is based on the polyhedral model [8], requiring code to be represented as a set of static affine loop nests[1]. In this work, we present a new theory of algorithmic species such that it is no longer polyhedral model based and thus broader applicable. Additionally, we introduce a more detailed SPECIES+ classification. We make the following contributions:

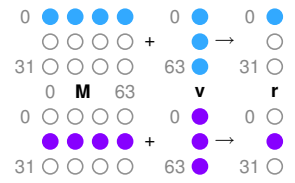
- We introduce a 5-tuple characterisation of array references with respect to loop nests (section III). On top of this, we define transformations to merge characterisations referring to the same array and to translate them into algorithmic species (section IV). This new theory allows us to classify non static affine loop nests (section V-B).
- We present SPECIES+, a classification based on more detailed 6-tuple abstractions that take the loop nest structure into account and retain additional performance-relevant information (section V-A).
- We describe a tool based on the presented theories to automatically classify program code (section VI).

II. Background and Motivation

Algorithmic species is an algorithm classification based on access patterns of arrays in loop nests [13]. The classification is designed to fulfil the following goals: 1) programmers can reason about their program code by means of algorithm classes, 2) performance models can use class information to predict performance, and 3) compilers can be designed based on the classification. To achieve this, the following requirements for classes are set [13]: automatically extracted, intuitive, formally defined, complete and fine-grained. Note that species are a means to *classify* parallel code rather than a tool to extract additional parallelism from program code.

To illustrate algorithmic species, let us consider the matrix-vector multiplication in figure 1. The corresponding species can be interpreted as follows: to produce a single element out of the total 32 elements in `r`, we need a chunk of data in the second dimension of `M` (a row) and the full array `v` of size 64. This captures the structure of the example and can be used for classification. More details and examples can be found in [13].

```
1 for (i=0; i<32; i++) {
2   r[i] = 0;
3   for (j=0; j<64; j++) {
4     r[i] += M[i][j]*v[j];
5   }
6 }
```



`M[0:31,0:63]chunk(-,0:63) ∧ v[0:63]full → r[0:31]element`

Fig. 1. Matrix-vector multiplication classified as species. The right hand side illustrates the address space of 2 iterations of the i-loop in different colours.

[1]We define static affine loop nests as loop nests with static loop control, affine loop bounds, affine conditional statements, and affine array references.

978-1-4799-1010-6/13/$31.00

Algorithmic species are used for example in a *skeleton*-based source-to-source compiler [14] to achieve (performance) portability across different architectures. Skeletons can be seen as parametrised template code for a specific class of computations on a specific target processor. The compiler's task is to instantiate such a skeleton and to generate efficient target code. In [14], skeletons correspond directly to the algorithmic species: choosing a skeleton is a matter of classifying the code.

Similar to related classifications such as array regions [5] and Æcute [11], algorithmic species give an abstraction of program code: information is lost in the translation from code to species. The level of abstraction of species is a trade-off to meet the aforementioned goals, e.g. it should be suitable for both manual and automatic uses. The main abstractions are:

- Species are specified with respect to loop nests for which the order of execution of nested loops and loop iterations is irrelevant, i.e. any order will preserve the semantics (e.g. loop i in figure 1).
- Species may give an over-approximation: not all specified accesses have to be made. For example, if the matrix-vector multiplication would not access element `V[19]` because of a condition guarding line 4 in figure 1, the species would remain unchanged.
- The performed computation is abstracted away. Species focus instead on parallelism and memory access patterns.

Because the algorithmic species theory [13] is based on the polyhedral model [8], completeness and automatic extraction can at best be achieved only for code that is represented as static affine loop nests. Therefore, we propose to develop a new formal definition of species that is not based on the polyhedral model. Furthermore, we propose to unify the original array access patterns (i.e. *element*, *chunk*, *neighbourhood*, *full*, and *shared*) to create a unified and more structured theory. The intuition behind this is that many of these patterns are special cases of others. For example, the *element* pattern can be seen as a *chunk* access pattern of size 1.

In addition to revisiting the theory of algorithmic species, we also introduce SPECIES+, a more detailed classification. This is motivated by the fact that two equal species could still have significantly different memory access patterns (shown in section VII). The reason for this is the abstraction of the order of execution of nested loops and loop iterations, limiting the identification of memory and cache-friendly access patterns. For example, a tiled loop and its non-tiled counterpart are currently classified as the same algorithmic species.

III. Characterising array references

We base our new theory for algorithmic species on characteristics of individual array references in loop nests. In this section we define this characterisation. For now, let us consider only static affine loop nests. We restrict our work to arrays that do not alias and assume functions to be inlined.

A. Basic case: single loop and 1-dimensional arrays

First, let us discuss the case for which our loop nest contains only a single loop, and array references are all 1-dimensional. Following, in section III-B, we consider the general case with a loop nest of one or more (N) loops and references to one or multi (M) dimensional arrays.

As an example of a single loop with 1-dimensional references, consider figure 2 and the reference to array `A` with respect to the i-loop. We characterise this reference by its name (A), its access type (r for read), its domain with respect to the loop (lower-bound 2 and upper-bound 7), its per-iteration accessed size (1 element), and its iteration step (1 element). This forms the 5-tuple $(A, r, [2..7], 1, 1)$. For the reference to `B`, we obtain in a similar way $(B, w, [0..5], 1, 1)$.

```
1 for(i=2; i<8; i++) {
2   B[i-2] = A[i];
3 }
```

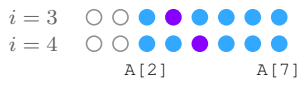


Fig. 2. Example code with a static affine loop. The right hand side illustrates the domain of `A` (filled circles) and two iterations of the i-loop (purple circles).

In general, we characterise each array reference in a static affine loop nest as a 5-tuple $\mathcal{R} = (\mathcal{N}, \mathcal{A}, \mathcal{D}, \mathcal{E}, \mathcal{S})$. The tuple's elements are defined as follows:

- $\mathcal{N}$ is the array's name, given as a string.
- $\mathcal{A} \in (r, w)$ is the access type (r for read and w for write).
- $\mathcal{D} \in [\mathbb{Z}..\mathbb{Z}]$ gives the integer address space domain of array references with respect to the loop nest, represented as an interval with a lower-bound and an upper-bound.
- $\mathcal{E} \in \mathbb{N}$ gives the number of elements accessed. Note that $\mathcal{E} = 1$ unless there is an additional loop inside the body of our reference loop nest (prior to merging: section IV).
- $\mathcal{S} \in (\mathbb{Z}, \mathbb{Q})$ gives the step. Note that the step can be negative in case of a backwards counting loop, zero in case of a loop independent reference, or a unit fraction in case of a non-monotonic step. For example, the fraction $\frac{1}{4}$ represents a step taken every 4 iterations.

To further illustrate the basic characterisation of array references, we discuss several other examples. Let us consider the code snippet of figure 3. The first loop (lines 1-7) and the second loop (lines 8-10) are functionally equivalent. For both loops, we find the characterisation $(G, r, [0..4], 1, 2)$ for the reference to `G` and $(H, w, [0..2], 1, 1)$ for the reference to `H`.

```
1 for(i=0; i<8; i++) {
2   if(i % 2 == 0) {
3     if(i < 6) {
4       H[i/2] = G[i];
5     }
6   }
7 }
8 for(i=0; i<3; i++) {
9   H[i] = G[i*2];
10 }
```

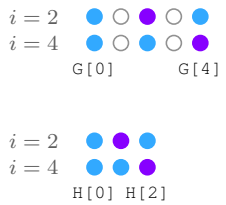


Fig. 3. Example of two functionally equal loops illustrating the characterisation. The illustrations of two iterations of the loop on the right hand side are valid for both loops. The `i/2` division is assumed to be an integer division.

Let us now consider the two functionally equivalent loops shown in figure 4. For these loops, we will construct the

references with respect to the outer i-loop only. Although the two code snippets are functionally equivalent, the characterisation of the references to array P is distinct. For the first loop (lines 1-6) we characterise the reference to P as $(P, r, [0..7], 2, 2)$, and to Q as $(Q, w, [0..3], 1, 1)$. Note that we obtain a step $\mathcal{S} = 2$ for P because of the j-loop. For the second loop (lines 7-9), we find two references to P, described as $(P, r, [0..6], 1, 2)$ and $(P, r, [1..7], 1, 2)$. The characterisation of Q remains as before. To obtain the same result for both loops, we describe the merging of two 5-tuples in section IV.

```
1 for(i=0; i<4; i++) {
2   Q[i] = 0;
3   for(j=0; j<2; j++) {
4     Q[i] += P[2*i+j];
5   }
6 }
7 for(i=0; i<4; i++) {
8   Q[i] = P[2*i]+P[2*i+1];
9 }
```

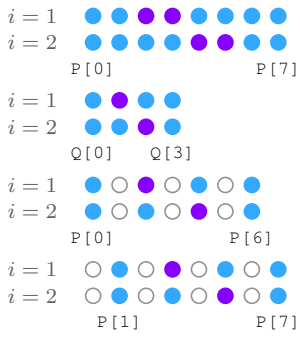


Fig. 4. Example of two functionally equal loops. The right hand side illustrates (top to bottom): the reference to P for the first loop (lines 1-6), the reference to Q for both loops, the first reference to P for the second loop (lines 7-9), and the second reference to P for the second loop.

B. General case: N loops and M-dimensional arrays

As discussed, the algorithmic species theory as presented in [13] abstracts away the execution order of the N loops and their iterations. Therefore, we perform the same abstraction in this section for our revised theory based on array reference characterisations. The more detailed classification SPECIES+ discussed in section V-A does not make this abstraction.

To be able to characterise M-dimensional arrays, we modify the 5-tuple $\mathcal{R}$ by adding a dimension to the domain $\mathcal{D}$, the number of elements $\mathcal{E}$, and the step $\mathcal{S}$. We represent the different array dimensions as a set with angular brackets, i.e. $\mathcal{D} = \langle \mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_M \rangle$, $\mathcal{E} = \langle \mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_M \rangle$, and $\mathcal{S} = \langle \mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_M \rangle$. The individual $\mathcal{D}_i$, $\mathcal{E}_i$, and $\mathcal{S}_i$ retain their definition of the 1-dimensional case (section III-A). The total number of elements now becomes the product of the number of elements in each dimension. In case there is only a single dimension, we omit the angular brackets from our notation.

Now, let us consider the characterisation of array references with respect to loops i and j for the example code in lines 1-5 of figure 5. If we apply our characterisation, we obtain the following tuples (along with their corresponding description as algorithmic species):

$$(R, r, \langle [0..7], [0..7] \rangle, \langle 1, 1 \rangle, \langle \frac{1}{8}, 1 \rangle) \rightarrow \texttt{R[0:7][0:7]element}$$
$$(S, w, [0..63], 1, 1) \rightarrow \texttt{S[0:63]element}$$
$$(T, r, [0..7], 1, \frac{1}{8}) \rightarrow \texttt{T[0:7]element}$$

In this characterisation, $\frac{1}{8}$ as a step for T represents a unit step every 8 loop iterations, and $\langle \frac{1}{8}, 1 \rangle$ as a step for R represents a unit step every 8 iterations in the first dimension, and a unit step in the second dimension (modulo the domain). The abstraction of algorithmic species allows us to rewrite lines 1-5 of figure 5 as lines 6-8. Although lines 6-8 have only a single loop, the characterisation of the references to arrays R, S, and T remains as above. We therefore re-classify this example using an extended characterisation in section V-A.

```
1 for(i=0; i<8; i++) {
2   for(j=0; j<8; j++) {
3     S[i*8+j] = R[i][j] + T[i];
4   }
5 }
6 for(k=0; k<64; i++) {
7   S[k] = R[k/8][k%8] + T[k/8];
8 }
```

Fig. 5. Example code with multiple loops and multi-dimensional arrays.

As a second example of multi-dimensional arrays, consider the matrix-vector multiplication of figure 1. If we characterise this code with respect to loop i, we obtain the following:

$$(M, r, \langle [0..31], [0..63] \rangle, \langle 1, 64 \rangle, \langle 1, 0 \rangle) \rightarrow \texttt{M[0:31,0:63]chunk}(-,\texttt{0:63})$$
$$(v, r, [0..63], 64, 0) \rightarrow \texttt{v[0:63]full}$$
$$(r, w, [0..31], 1, 1) \rightarrow \texttt{r[0:31]element}$$

Here, the step $\langle 1, 0 \rangle$ for M reflects the fact that references to the second dimension are independent of the i-loop: a whole row of the matrix is accessed at every iteration of the i-loop.

IV. Merging and translating into species

Before being able to translate array reference characterisations into algorithmic species, we need to perform a merging step. We merge the characterisations of individual array references in order to: 1) form compound patterns such as *tile* and *neighbourhood*, and 2) abstract away implementation choices (e.g. consider the loop unrolling performed in figure 4).

A. Merging array references

For a pair of array references $\mathcal{R}_a$ and $\mathcal{R}_b$, merging is only performed for references with equal names ($\mathcal{N}_a = \mathcal{N}_b$), access types ($\mathcal{A}_a = \mathcal{A}_b$) and steps ($\mathcal{S}_a = \mathcal{S}_b$). Let us first consider an example of merging. In the example of figure 4, the references to array P in the second loop (lines 7-9) can be merged. For this example, we merge $(P, r, [0..6], 1, 2)$ and $(P, r, [1..7], 1, 2)$ into $(P, r, [0..7], 2, 2)$. This gives us the same result as the characterisation of P for the first loop of figure 4 (lines 1-6).

We describe the rules for merging in the form of algorithm 1. The algorithm is repeatedly executed until no changes to the set of array references R for a single loop nest are made. It considers each pair $\mathcal{R}_a, \mathcal{R}_b \in R$ and proceeds as follows:

1) Check the condition of matching names, access type, and step (line 2 of algorithm 1).
2) Check whether the lengths of the domains are equal and if there is intersection (line 3).

3) Calculate the domain $\mathcal{D}_{new}$ as the union $\mathcal{D}_a \cup \mathcal{D}_b$ and the number of elements $\mathcal{E}_{new}$ as the absolute difference between the bounds of the domains (lines 4-5).
4) Continue only if the new number of elements $\mathcal{E}_{new}$ is within a threshold t_{gap} (illustrated later on) from the sum of the old number of elements (line 6).
5) Replace the tuples $\mathcal{R}_a$ and $\mathcal{R}_b$ with $\mathcal{R}_{new}$ (lines 7-8).

Input: array references R (w.r.t. a loop nest)
1 **foreach** $\{\mathcal{R}_a, \mathcal{R}_b\} \in R$ **do**
2 **if** $\mathcal{N}_a = \mathcal{N}_b$ **and** $\mathcal{A}_a = \mathcal{A}_b$ **and** $\mathcal{S}_a = \mathcal{S}_b$ **then**
3 **if** $|\mathcal{D}_a| = |\mathcal{D}_b|$ **and** $\mathcal{D}_a \cap \mathcal{D}_b \neq \emptyset$ **then**
4 $\mathcal{D}_{new} = \mathcal{D}_a \cup \mathcal{D}_b$
5 $\mathcal{E}_{new} = |min(\mathcal{D}_a) - min(\mathcal{D}_b)|$
6 **if** $\mathcal{E}_a + \mathcal{E}_b + t_{gap} > \mathcal{E}_{new}$ **then**
7 $\mathcal{R}_{new} = (\mathcal{N}_a, \mathcal{A}_a, \mathcal{D}_{new}, \mathcal{E}_{new}, \mathcal{S}_a)$
8 replace $\mathcal{R}_a$ and $\mathcal{R}_b$ with $\mathcal{R}_{new}$ in R
9 **end**
10 **end**
11 **end**
12 **end**

Algorithm 1: Algorithm to perform the merging of a pair of array references.

To illustrate merging in case of a *neighbourhood* access pattern, consider the example of figure 6. Here, we find 3 references to array `V` which we characterise as $\mathcal{R}_{\texttt{V[i-1]}} = (V, r, [0..5], 1, 1)$, $\mathcal{R}_{\texttt{V[i]}} = (V, r, [1..6], 1, 1)$, and $\mathcal{R}_{\texttt{V[i+1]}} = (V, r, [2..7], 1, 1)$. By calculating the combined domain $\mathcal{D}$ and the number of elements $\mathcal{E}$, we obtain $(V, r, [0..7], 3, 1)$. This captures the overlap of references between iterations as the number of elements $\mathcal{E}$ (3) is now larger than the step $\mathcal{S}$ (1).

```
1 for(i=1; i<7; i++) {
2   W[i] = V[i-1] +
3          V[i] +
4          V[i+1];
5 }
```

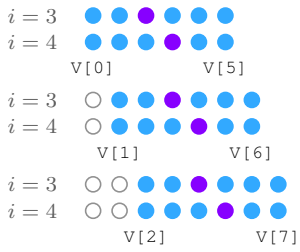


Fig. 6. Example of a *neighbourhood* read access pattern. The right hand side illustrates two iterations of the i-loop for the 3 references to `V`.

Furthermore, let us consider the example of interpolation (figure 7) where the set of reads per iteration is not convex: there is a gap between references `K[i-1]` and `K[i+1]`. For the input array references, we find $(K, r, [0..4], 1, 2)$ and $(K, r, [2..6], 1, 2)$. Now, we can choose to treat these as two separate *element* accesses, or merge them into a *neighbourhood* access. If we perform the latter, we obtain $(K, r, [0..6], 3, 2)$, which gives us an over-approximation: we may access 3 elements each iteration, but do access only the 2 extreme elements. Whether or not such an outcome is desired can be set with the t_{gap} variable in algorithm 1. The original algorithmic species theory [13] does not discuss the issue of non-convex sets, implying $t_{gap} = \infty$.

```
1 for(i=1; i<6; i+=2) {
2   K[i] = K[i-1] + K[i+1];
3 }
```

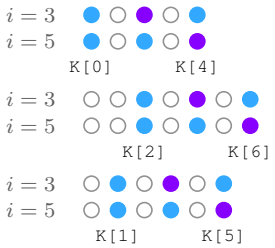


Fig. 7. Example of an implementation of interpolation. The two read references to array `K` are candidates for merging.

B. Translating array references into species

Once the found array references are merged (where possible), they can be translated into the algorithmic species of [13]. Species can thus be seen as an abstract representation of a combination of our 5-tuple characterisations. For automated use (e.g. in compilers), it might be advantageous to use the characterisations directly, as they could provide additional information. Nevertheless, in case of manual uses, using species can provide better understandability and usability.

Input: array references R after merging (w.r.t. a loop nest)
1 X = $\emptyset$
2 **foreach** $\mathcal{R}_a \in R$ **do**
3 **if** $\mathcal{S}_a = 0$ **and** $\mathcal{A}_a = r$ **then**
4 X $\leftarrow \mathcal{N}_a$ $\mathcal{D}_a$ full
5 **else if** $\mathcal{S}_a = 0$ **and** $\mathcal{A}_a = w$ **then**
6 X $\leftarrow \mathcal{N}_a$ $\mathcal{D}_a$ shared
7 **else if** $\mathcal{E}_a = 1$ **then**
8 X $\leftarrow \mathcal{N}_a$ $\mathcal{D}_a$ element
9 **else if** $\mathcal{S}_a < \mathcal{E}_a$ **then**
10 X $\leftarrow \mathcal{N}_a$ $\mathcal{D}_a$ neighbourhood $(\mathcal{E}_a)$
11 **else**
12 X $\leftarrow \mathcal{N}_a$ $\mathcal{D}_a$ chunk $(\mathcal{E}_a)$
13 **end**
14 **end**

Algorithm 2: Algorithm to extract patterns from array references.

We present algorithm 2 to extract patterns from our characterisation. The algorithm processes each characterisation $\mathcal{R}_a$ (after merging) as follows (line numbers refer to algorithm 2):

- **3-6** If $\mathcal{R}_a$ has a zero step, it belongs either to the *full* (for reads) or *shared* (for writes) pattern.
- **7-8** Else, if precisely a single element of $\mathcal{R}_a$ is accessed every iteration, it is classified as the *element* pattern.
- **9-10** Else, if the amount of elements accessed is larger than the step size, there is overlap between iterations. This is captured by the *neighbourhood* pattern.
- **11-13** If non of the above holds, $\mathcal{R}_a$ belongs to the *chunk* pattern. This is the case if multiple elements are accessed, but there is no overlap between successive iterations.

As shown in algorithm 2, the algorithm also prefixes the names ($\mathcal{N}_a$) and domains ($\mathcal{D}_a$), and includes the number of elements accessed for the *neighbourhood* and *chunk* patterns. The final algorithmic species is obtained by taking the results of algorithm 2 (in X) and combining them as follows:

$I_1 \wedge ... \wedge I_n \rightarrow O_1 \wedge ... \wedge O_m$, in which I_x represent inputs ($\mathcal{A}_x = r$) and O_x represent outputs ($\mathcal{A}_x = w$).

V. Beyond algorithmic species

One of the goals of the new theory behind algorithmic species is to be able to extend the applicability and expressiveness of species. We do this in two ways: 1) we provide SPECIES+, a more detailed classification, and 2) we classify non static affine loop nests.

A. SPECIES+: *a more detailed classification*

Embedding additional information in our array reference characterisations can improve the current uses of algorithmic species and enable new uses. For example, consider the three references to the array `X` in figure 8. All three references would be characterised as $(X, r, [0..63], 1, 1)$ with respect to the loop nest, while cache behaviour (e.g. row-major or column-major) and performance can differ significantly.

```
1 for(i=0; i<8; i++) {
2   for(j=0; j<8; j++) {
3     Y[i][j] = X[i*8+j] + X[j*8+i];
4   }
5 }
6 for(k=0; k<64; k++) {
7   Z[k] = X[k];
8 }
```

Fig. 8. Accessing the 64 first elements of array `X` in three different ways.

In this section we propose to create a more detailed characterisation, allowing us to distinguish among others the accesses made to `X` in figure 8. To do so, we modify the 5-tuple array reference characterisation to obtain a 6-tuple by appending a repetition factor $\mathcal{X}$. Earlier, we included the dimensions of the arrays into $\mathcal{S}$ to obtain $\mathcal{S} = \langle \mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_M \rangle$. Now, we modify each step $\mathcal{S}_x$ to become a set of size N (with N the number of loops in the nest). To distinguish the array dimensions from the number of loops, we use a different notation: $\mathcal{S}_x = \mathcal{S}_{x,1}|\mathcal{S}_{x,2}|...|\mathcal{S}_{x,N}$. This can also be represented as a matrix with N columns and M rows:

$$\begin{bmatrix} \mathcal{S}_{1,1} & \mathcal{S}_{2,1} & ... & \mathcal{S}_{N,1} \\ \mathcal{S}_{1,2} & \mathcal{S}_{2,2} & ... & \mathcal{S}_{N,2} \\ ... & ... & & ... \\ \mathcal{S}_{1,M} & \mathcal{S}_{2,M} & ... & \mathcal{S}_{N,M} \end{bmatrix}$$

The new repetition factor $\mathcal{X}$ reflects the iteration count of the two loops: it is also a set of N items and uses the same notation: $\mathcal{X} = \mathcal{X}_1|\mathcal{X}_2|...|\mathcal{X}_N$ We use the name SPECIES+ to refer to a set of 6-tuple classifications for a given loop nest.

For example, consider the access to array `A` in figure 9. The new 6-tuple array reference characterisation becomes $(A, r, \langle [0..7][0..7] \rangle, \langle 1, 1 \rangle, \langle 1|0, 0|1 \rangle, 8|8)$. Here, the step $\langle 1|0, 0|1 \rangle$ or $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ represents a reference to the first dimension every iteration of the first loop i and a reference to the second dimension every iteration of the second loop j. The repetition factor $8|8$ represents the iteration count of the two loops. For the access to array `B` in figure 9 the step changes into $\langle 0|1, 1|0 \rangle$. Finally, for `C`, the step becomes $\langle 1, 1 \rangle$ and the repetition factor becomes 8 (there is only a single loop).

```
1 for(i=0; i<8; i++) {
2   for(j=0; j<8; j++) {
3     B[j][i] = A[i][j];
4   }
5 }
6 for(i=0; i<8; i++) {
7   C[i][i] = 0;
8 }
```

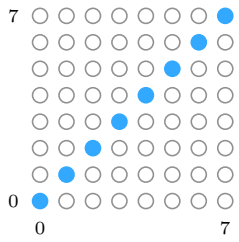


Fig. 9. Additional examples of two-dimensional array references. The right hand side illustrates the accesses made to the array `C`.

Let us consider the example of `X` in figure 8 again. Now, we can construct the more detailed 6-tuple array reference characterisations as follows:

$$\mathcal{R}_{\texttt{X[i*8+j]}} = (X, r, [0..63], 1, 8|1, 8|8)$$
$$\mathcal{R}_{\texttt{X[j*8+i]}} = (X, r, [0..63], 1, 1|8, 8|8)$$
$$\mathcal{R}_{\texttt{X[k]}} = (X, r, [0..63], 1, 1, 64)$$

The different steps $\mathcal{S}$ now show the differences between the 3 access types. We are now able to distinguish between e.g. a row-major and a column-major access. For completeness, let us also re-classify the accesses made to `R`, `S`, and `T` as found in example 5 using SPECIES+:

$$\mathcal{R}_{\texttt{R[i][j]}} = (R, r, \langle [0..7][0..7] \rangle, \langle 1, 1 \rangle, \langle 1|0, 0|1 \rangle, 8|8)$$
$$\mathcal{R}_{\texttt{R[k/8][k\%8]}} = (R, r, \langle [0..7][0..7] \rangle, \langle 1, 1 \rangle, \langle \frac{1}{8}, 1 \rangle, 64)$$
$$\mathcal{R}_{\texttt{S[i*8+j]}} = (S, w, [0..63], 1, 8|1, 8|8)$$
$$\mathcal{R}_{\texttt{S[k]}} = (S, w, [0..63], 1, 1, 64)$$
$$\mathcal{R}_{\texttt{T[i]}} = (T, r, [0..7], 1, 1|0, 8|8)$$
$$\mathcal{R}_{\texttt{T[k/8]}} = (T, r, [0..7], 1, \frac{1}{8}, 64)$$

B. *Non static affine loop nests*

So far, we have considered only static affine loop nests. In this section, we discuss a number of examples that violate one of the constraints of static affine loop nests: 1) non-static loop control, 2) non-affine loop bounds, 3) non-affine conditional statements, and 4) non-affine array references. The examples are listed in figure 10. For simplicity, we use the original 5-tuple representation in this section instead of SPECIES+.

```
1 // Non-static control
2 i = 0;
3 while(i<8) {
4   B[i] = A[i];
5   i = i + A[i];
6 }
7 // Non-affine bound
8 for(i=0;i<8-i*i;i++) {
9   H[0] = G[i];
10 }
```

```
11 // Non-affine condition
12 for(i=0; i<8; i++) {
13   if (P[i] > 12) {
14     P[i] = 0;
15   }
16 }
17 // Non-affine references
18 for(i=0; i<8; i++) {
19   S[T[i]] = R[i*i];
20 }
```

Fig. 10. Examples of non static affine loop nests.

Consider the example with non-static control in lines 1-6 of figure 10. Every iteration of the loop, we increment i by

a value dependent on the memory state. This leads to a case where the iteration step $\mathcal{S}$ is unknown at compile-time and might not even be constant. In this case, we do not characterise the references.

Now, let us consider the example with non-affine loop bounds (lines 7-10, figure 10). Although the loop bound is not affine, we can still find the upper-bound ($i \leq 3$) at compile-time for this particular example. Doing so, we obtain $(G, r, [0..3], 1, 1)$ and $(H, w, [0..0], 1, 0)$. However, determining the domain $\mathcal{D}$ for a loop with non-affine loop bounds is not always possible. Consider the loop bound `i<G[i]` instead. In this case, we can only provide an over-approximation of the domain based on the programmer's knowledge or on the type of `G`, e.g. 255 for a 1-byte `unsigned char` C data-type.

In lines 11-16 of figure 10 we show an example with a non-affine conditional statement. In case the condition would not be present, the reference to `P` would be characterised as $(P, w, [0..7], 1, 1)$. Since we cannot know upfront whether or not the condition will evaluate to true or false, we have to use the original classification. This can be seen as an over-approximation: stepping through the domain with a unit step, but not necessarily performing a read access every time.

The final example in lines 17-20 of figure 10 shows an affine array reference `T` and two non-affine array references `R` and `S`. We characterise the affine reference as $(T, r, [0..7], 1, 1)$. The reference to `R` has a non-constant step. We therefore use $(R, r, [0..49], 1, 1)$, giving an over-approximation of the domain and the step: not all elements are accessed. For the reference to `S`, we could include type information of `T`. Assuming a range of 0 to 255, we obtain $(S, w, [0..255], 256, 0)$. This is as if we write to all locations every loop iteration.

VI. Automatic extraction of classes

The original algorithmic species theory [13] included ASET, the polyhedral-based 'algorithmic species extraction tool' [6]. Along with the new theory, we present a new automatic extraction tool named A-DARWIN[2]. This tool is largely equal to ASET in terms of functionality, but is different internally. The tool is based on a C99 parser which allows analysis on an abstract syntax tree (AST). From the AST, the tool extracts the array references and constructs a 5 or 6-tuple $\mathcal{R}$ for each loop nest. Following, the merging is applied as described by algorithm 1 and the species are extracted as described by algorithm 2. Finally, the species are inserted as pragma's in the original source code.

Algorithmic species and A-DARWIN are not designed to perform loop transformations nor to create parallel loops. A parallelising tool (such as PLUTO) could be used as a pre-processor to create (e.g. through loop skewing) and identify parallel loops. In case parallel loops are not identified a-priori, A-DARWIN also includes basic dependence analysis. The main reason to include this is to allow A-DARWIN to be usable as a stand-alone tool. First, we apply Bernstein's conditions, leaving us with pairs $\mathcal{R}_a$ and $\mathcal{R}_b$ for which $\mathcal{N}_a = \mathcal{N}_b$ and $\mathcal{A}_a \neq r \vee \mathcal{A}_b \neq r$. We apply a combination of the GCD and Banerjee tests [12] on the remaining pairs. Together, these tests are conservative, i.e. we might not find all parallel loops. This can be improved by invoking more advanced tests, such as the I-test [12] or the Omega test [15].

Static analysis has a limited scope of applicability. For example, loop nests such as those in figure 10 cannot always be fully analysed. Therefore, A-DARWIN will classify species in some cases as over-approximations. These over-approximations can be tightened using either programmer's knowledge (manual) or run-time information (dynamic).

Since the 6-tuple classifications SPECIES+ can embed more information than the original algorithmic species, A-DARWIN also contains an option to output the full SPECIES+ classification instead.

VII. Evaluation

We have evaluated A-DARWIN and the presented theory on the PolyBench benchmark suite, obtaining the same results as in [13]. We do not further discuss these results here. However, since we also provide the alternative SPECIES+ classification, we discuss a few examples in this section to highlight the classification differences when compared to algorithmic species. Additionally, we illustrate a nested classification and discuss the known limitations to the current theory. For an evaluation of the use of species within a compiler, we refer to [14].

Let us first consider the `syrk` example from the PolyBench suite as given in figure 11. If we classify this example as algorithmic species with respect to loops i and j, we find an *element* access pattern for `B` both as input and as output. Furthermore, we find `A` twice as input with the *chunk* pattern, in both cases accessing a full row. In the algorithmic species classification, it is not possible to make a distinction between the two accesses to `A`. In contrast, if we apply the 6-tuple classification SPECIES+ to `A`, we obtain the following:

$$\mathcal{R}_{\texttt{A[i][k]}} = (A, r, \langle [0..N][0..M] \rangle, \langle 1, M \rangle, \langle 1|0, 0|0 \rangle, N|N)$$
$$\mathcal{R}_{\texttt{A[j][k]}} = (A, r, \langle [0..N][0..M] \rangle, \langle 1, M \rangle, \langle 0|1, 0|0 \rangle, N|N)$$

This does not only show the difference between the two references (a step for the i loop or for the j loop), but also shows the fact that every row is accessed N times.

```
1 for (i=0; i<=N; i++)
2   for (j=0; j<=N; j++) {
3     B[i][j] *= beta;
4     for (k=0; k<=M; k++)
5       B[i][j] += alpha * A[i][k] * A[j][k];
6   }
```

```
B[0:N,0:N]|element ∧ A[0:N,0:M]|chunk(-,0:M) ∧
A[0:N,0:M]|chunk(-,0:M) → B[0:N,0:N]|element
```

Fig. 11. Code to compute symmetric rank-k operations (`syrk` in the PolyBench suite) and its classification as algorithmic species.

As another example, consider the code in figure 12, which shows a backwards counting loop and two read references

[2] A-DARWIN is short for 'automatic Darwin', named after the author of *'On the Origin of Species'*. It is available through http://parse.ele.tue.nl/species/.

to C. With algorithmic species, we are unable to distinguish the two reads. However, if we use the 6-tuple classification instead, we obtain:

$$\mathcal{R}_{\texttt{C[i]}} = (C, r, [0..7], 1, -1, 8)$$
$$\mathcal{R}_{\texttt{C[7-i]}} = (C, r, [0..7], 1, 1, 8)$$

The difference between the two steps (-1 and 1) shows the order in which the array C is referred to, but also the relation between the two references.

```
1 for (i=7; i>=0; i--) {
2   D[i] = C[i] + C[7-i];
3 }
```

```
C[0:7]|element ∧ C[0:7]|element → C[0:7]|element
```

Fig. 12. Code example of a backwards counting loop with two reads to C, and its classification as algorithmic species.

Next, let us consider an example of loop tiling. Loop tiling is a loop transformation that can be used for example to obtain better cache behaviour. In figure 13, we show an example of non-tiled code (lines 1-3) and tiled code (lines 4-8). In this case, the tile size is 2x2, visible through the step-size of the i and j loops and the bounds of the ii and jj loops. The reference to E would classify as the species '`E[0:7][0:7]element`' in both cases. However, with SPECIES+, we do capture the difference:

original: $(E, w, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1|0, 0|1\rangle, 8|8)$

tiled: $(E, w, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 2|0|1|0, 0|2|0|1\rangle, 4|4|2|2)$

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

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

Fig. 13. Example code (left hand side) and a tiled version (right hand side).

Additionally, we illustrate the characterisation of array references at different loop levels and for individual statements. In figure 14 we show the matrix-vector multiplication of figure 1 again, but now with in-lined 6-tuple characterisations at different levels. Note that the outer-most classification is with respect to the i-loop only. Furthermore, note that because the step $\mathcal{S}$ and repetition $\mathcal{X}$ take their (inner) dimensionality from the number of loops considered, characterisations can include the empty set ($\emptyset$). The example shows that the theory is not limited to loops, but can also be applied to individual statements. Furthermore, it illustrates the isolation of the loop body: although array r is read and written in the loop body, it is classified from the outer-loop perspective as write only because all reads to individual elements occur after writes.

Finally, we discuss a number of examples that illustrate the limitations of algorithmic species and the 6-tuple SPECIES+ classification. First, let us consider the write access to a triangular matrix F in lines 1-6 of figure 15. Because

```
1  // R_r = (r, w, [0..31], 1, 1, 32)
2  // R_M = (M, r, ⟨[0..31][0..63]⟩, ⟨1, 64⟩, ⟨1, 0⟩, 32)
3  // R_v = (v, r, [0..63], 1, 0, 32)
4  for (i=0; i<32; i++) {
5    // R_r = (r, w, [i..i], 1, ∅, ∅)
6    r[i] = 0;
7    // R_r = (r, w, [i..i], 1, 0, 64)
8    // R_r = (r, r, [i..i], 1, 0, 64)
9    // R_M = (M, r, ⟨[i..i][0..63]⟩, ⟨1, 1⟩, ⟨0, 1⟩, 64)
10   // R_v = (v, r, [0..63], 1, 1, 64)
11   for (j=0; j<64; j++) {
12     // R_r = (r, w, [i..i], 1, ∅, ∅)
13     // R_r = (r, r, [i..i], 1, ∅, ∅)
14     // R_M = (M, r, ⟨[i..i][j..j]⟩, ⟨1, 1⟩, ⟨∅, ∅⟩, ∅)
15     // R_v = (v, r, [j..j], 1, ∅, ∅)
16     r[i] += M[i][j]*v[j];
17   }
18 }
```

Fig. 14. Matrix-vector multiplication with per-statement characterisations in-lined (lines starting with //), referring to the following loop or statement.

the j-loop's bounds change every iteration of the i-loop, we will have to characterise F as an over-approximation: $(F, w, \langle[0..7][0..7]\rangle, \langle 1, 1\rangle, \langle 1|0, 0|1\rangle, 8|8)$. The exact iteration domain can be described as a polyhedron, as is done in e.g. [5].

```
1 // Triangular matrix
2 for(i=0; i<8; i++) {
3   for(j=i; j<8; j++) {
4     F[i][j] = 0;
5   }
6 }
```

```
7  // Random references
8  for(i=0; i<8; i++)
9    G[rand()%8] = 0;
10 // Fractional indexing
11 for(i=0; i<8; i++)
12   H[(i+1)/4] = 0;
```

Fig. 15. Additional examples of loop nests to illustrate the limitations.

As a second example, consider lines 7-9 of figure 15. Here, we show a random reference independent of the iterator i to G. With our 6-tuple characterisation, we obtain $(G, w, [0..7], 8, 0, 8)$, as if we are referencing to the entire array G every iteration. From the loop nest, we do know however that we are referencing only one element per iteration.

As a final example, we show fractional indexing with an offset in lines 10-12 of figure 15. Although we are able to represent H[i/4] with a step of $\frac{1}{4}$, we are not able to represent the offset in H[(i+1)/4]. As an extension, we could allow the domain to include fractions for such non affine cases. This would give us $(H, w, [\frac{1}{4}..2], 1, \frac{1}{4}, 8)$.

VIII. Related work

We discuss related work in this section. We refer to [13] for a comparison of the original algorithmic species with other classifications such as skeletons and Berkeley's dwarfs.

Related to our 5 or 6-tuple description is the theory of *convex array regions* [5]. Array regions summarise the memory accesses (reads and writes) performed by a function, a loop, or one or more statements. For example, the elements that are read during a set of statements s with memory state σ for the complete example in figure 2 can be described as $\mathcal{R}(s, \sigma) = \{A[\phi_1] | 2 \leq \phi_1 \leq 7\}$. Here, σ is not used. It is used for example for describing the reads for the same example's

loop body: $\mathcal{R}(s,\sigma) = \{A[\phi_1] | \phi_1 = \sigma(i)\}$. Array regions are described as a convex polyhedron and are thus abstractions of program code, similar to our domain description $\mathcal{D}$. In contrast to our work, array regions do not describe the complete access pattern, but merely capture our tuple's name $\mathcal{N}$, direction $\mathcal{A}$, and domain $\mathcal{D}$. Array regions are used for example to perform loop fusion and fission, dependence analysis, and data transfer optimisations [10].

The Array-OL specification language [2] has similarities to our 6-tuple representation. Array-OL models code at multiple levels. First of all, there is a task level representing loop nests as communicating tasks. An individual task is then repeated according to Array-OL's $s_{repetition}$. At each repetition a task accesses data by *patterns*. Each pattern is characterised by a paving matrix (P), a fitting matrix (F) and a pattern shape ($s_{pattern}$). Furthermore, Array-OL provides an origin vector ($\mathbf{o}$) and the array size (s_{array}). Array-OL's $s_{repetition}$ and P are closely related to our repetition factor $\mathcal{X}$ and step $\mathcal{S}$ respectively. The patterns are abstracted in our representation as the number of elements $\mathcal{E}$. In contrast to our work, Array-OL cannot be applied to non static affine loop nests and is not as suitable for classification purposes.

We briefly describe other related classifications which work on a different abstraction level. This includes the representation of loop nests in the polyhedral model, such as the representation used in the integer set library *isl* [16]. In *isl*, iterations of a loop nest are represented as integer points in a polytope using first order logic. In other work, polyhedral process networks [17] are introduced to provide a higher-level polyhedral-based classification of program code. Furthermore, the programming model Æcute [11] creates a decoupled access/execute specifications of program code. The language PENCIL [1] allows programmers or compilers to create summary functions to describe array references. Finally, the basic classification 'idioms' [3] provides a tool for automatic extraction from program code.

Alternatively, classifications can be based on dynamic information. For example, [4] uses performance counters to determine compiler optimisation settings. Although such work is targeted at a very specific goal, performance counter features can also be seen as a classification of code. In this way, classes capture many detailed program code characteristics. However, this also has drawbacks: manual or static extraction of classes is not possible, classes are not intuitive, the classification is architecture specific and classes are too fine-grained (e.g. not all performance counters are relevant).

IX. Conclusions and future work

In this work we have introduced a new technique to classify array references in loop nests as 5-tuple array reference characterisations. We have shown that these characterisations (or *summaries*) can be merged and transformed into *algorithmic species* [13], an earlier classification of loop nests. In comparison to the polyhedral-based algorithmic species theory, we are no longer limited to static affine loop nests. Additionally, we extended the theory to obtain the more detailed 6-tuple SPECIES+ classification, potentially improving the current uses of algorithmic species or enabling new uses. Furthermore, we have introduced a tool to automatically classify program code.

With the new theory, tool and SPECIES+, we have set a basis to address the challenges of programming, performance prediction, and code generation for parallel processor architectures. Through compilers such as [14], our architecture-agnostic classification can enable performance fine-tuning and achieve (performance) portability.

As part of future work, we plan to investigate the possibilities of loop fusion and fission at the level of species using the new 6-tuple SPECIES+ characterisations. Moreover, inspired by [5] and our merge operator, we plan to investigate the possibilities of describing other transformations or analysis passes in the form of operations on the SPECIES+ classification.

References

[1] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. In *WOLFHPC*, 2012.

[2] P. Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical Report RR-6113, INRIA, 2007.

[3] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *ICS '11: International Conference on Supercomputing*. ACM, 2011.

[4] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *CGO '07: Code Generation and Optimization*. IEEE, 2007.

[5] B. Creusillet and F. Irigoin. Exact versus Approximate Array Region Analyses. In *LCPC '97: Languages and Compilers for Parallel Computing*. Springer, 1997.

[6] P. Custers. Algorithmic Species: Classifying Program Code for Parallel Computing. Master's thesis, Eindhoven University of Technology, 2012.

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA '11: 38th International Symposium on Computer Architecture*. ACM, 2011.

[8] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Springer International Journal of Parallel Programming*, 20:23–53, 1991.

[9] S. H. Fuller and L. I. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44:31–38, 2011.

[10] S. Guelton, M. Amini, and B. Creusillet. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In *LCPC '12: Languages and Compilers for Parallel Computing*. Springer, 2012.

[11] L. Howes, A. Lokhmotov, A. Donaldson, and P. Kelly. Deriving Efficient Data Movement from Decoupled Access/Execute Specifications. In *HiPEAC '09: High Performance Embedded Architectures and Compilers*. Springer, 2009.

[12] X. Kong, D. Klappholz, and K. Psarris. The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):342–349, 1991.

[13] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming. *ACM TACO: Transactions on Architecture and Code Optimisations*, 9(4):Article 40, 2013.

[14] C. Nugteren, P. Custers, and H. Corporaal. Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification. In *APPT '13: Advanced Parallel Processing Technology*. Springer, 2013.

[15] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*. ACM, 1991.

[16] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In *ICMS 2010: International Conference on Mathematical Software*. Springer, 2010.

[17] S. Verdoolaege. Polyhedral Process Networks. *Handbook of Signal Processing Systems*, pages 931–965, 2010.

[18] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Arch. News*, 23(1):20–24, 1995.