Algorithmic Species: A Classification of Affine Loop Nests for Parallel Programming

CEDRIC NUGTEREN, PIETER CUSTERS, and HENK CORPORAAL, Eindhoven University of Technology

Code generation and programming have become ever more challenging over the last decade due to the shift towards parallel processing. Emerging processor architectures such as multi-cores and GPUs exploit increasingly parallelism, requiring programmers and compilers to deal with aspects such as threading, concurrency, synchronization, and complex memory partitioning. We advocate that programmers and compilers can greatly benefit from a structured classification of program code. Such a classification can help programmers to find opportunities for parallelization, reason about their code, and interact with other programmers. Similarly, parallelising compilers and source-to-source compilers can take threading and optimization decisions based on the same classification. In this work, we introduce *algorithmic species*, a classification of affine loop nests based on the polyhedral model and targeted for both automatic and manual use. Individual classes capture information such as the structure of parallelism and the data reuse. To make the classification applicable for manual use, a basic vocabulary forms the base for the creation of a set of intuitive classes. To demonstrate the use of algorithmic species, we identify 115 classes in a benchmark set. Additionally, we demonstrate the suitability of algorithmic species for automated uses by showing a tool to automatically extract species from program code, a species-based source-to-source compiler, and a species-based performance prediction model.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures; D.3.3 [Programming Languages]: Miscellaneous

General Terms: Performance, Languages

Additional Key Words and Phrases: Algorithm classification, parallel programming, polyhedral model

ACM Reference Format:

Nugteren, C., Custers, P., and Corporaal, H. 2013. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Architect. Code Optim.* 9, 4, Article 40 (January 2013), 25 pages. DOI = 10.1145/2400682.2400699 <http://doi.acm.org/10.1145/2400682.2400699>

1. INTRODUCTION

Single-processor performance has shown an exponential growth over the past decades, enabling technology to play a crucial role in our daily life. This exponential growth ended in 2004, limited mainly by power dissipation problems [Fuller and Millett 2011]. Performance growth was re-enabled through parallelism, i.e. placing multiple processor cores per chip. Accelerators such as the Graphics Processing Unit (GPU) and the Intel Many Integrated Core (MIC) are examples of massively parallel processors, benefiting from tens to thousands of small programmable processing cores.

The shift towards parallel processing has also created an increasingly heterogeneous computing environment, exploiting accelerators to off-load computations. Both

Authors' address: C. Nugteren (Corresponding author), P. Custers, and H. Corporaal, Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands; email: C.nugteren@tue.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART40 \$15.00

DOI 10.1145/2400682.2400699 <http://doi.acm.org/10.1145/2400682.2400699>

parallelism and heterogeneity have made programming a challenging task: programmers are faced with a variety of new programming languages and are required to have detailed architectural knowledge in order to optimize their applications for their target processor [Ryoo et al. 2008]. Apart from programming, issues such as maintainability, functional portability, and performance portability have become of major importance. Solving such issues using automation has become an important goal by compiler developers and researchers [Fuller and Millett 2011]. Areas currently investigated include new programming frameworks, language extensions, libraries, optimizing and auto-tuning compilers, and auto-parallelizing source-to-source compilers.

In this work we argue that programmers and compilers targeting parallel and heterogeneous systems can greatly benefit from a structured representation of program code, i.e. a set of rules determining the code's *class* based on a given set of properties. Such a structured representation in the form of an algorithm classification can help solve the aforementioned programming, maintainability, and portability issues. Examples of existing algorithm classifications are Berkeley *dwarfs* [Asanovic et al. 2009], algorithmic skeletons [Cole 1991], and the Galois classification system [Pingali et al. 2011]. In this work we introduce *algorithmic species*, a new algorithm classification based on the polyhedral model [Pouchet et al. 2007] which fulfils the following goals: (1) programmers targeting parallel processors will be able to reason about their program code by means of algorithm classes, and (2) compiler engineers and researchers will be able to design their compilers based on this classification. In a way, algorithmic species can be seen as transforming polyhedral information into an algorithm classification. The major contributions of this work are summarized as follows.

- We introduce *algorithmic species* in Section 4. Algorithmic species is an algorithm classification which fulfils the aforementioned goals. The individual classes are formally defined, easy to understand, applicable to any affine loop nest, and describe program code in a large amount of detail, capturing the structure of parallelism, opportunities for data reuse, locality information, and data sizes.
- We evaluate both existing classifications (in Section 3) and our classification (in Section 6) in great detail, assessing the quality with respect to five requirements (defined in Section 2). When comparing our classification with others, we conclude that algorithmic species is the only classification which fulfils the requirements and is able to meet the aforementioned goals.
- We demonstrate the applicability of the work in Section 5. We present and discuss the results of identifying 115 species in a benchmark set, capturing all loops with parallelism. We furthermore identify the use of algorithmic species in a number of tools and models: (1) a tool to extract species automatically from C code, (2) a skeleton-based source-to-source compiler, and (3) a species-based performance prediction model.

2. MOTIVATION

The shift towards a heterogeneous and parallel computing environment has made programming and compilation challenging tasks: exploiting multi-threading and efficiently using a processor's memory hierarchy are two examples of challenges faced by programmers (looking for a manual solution) and compilers (looking for an automated solution). We advocate that an algorithm classification can greatly alleviate such tasks for programmers (manual uses) and compilers (automated uses).

First, we envisage programmers to use an algorithm classification as a tool to ease parallel programming. Such a tool can help programmers to identify problems common to algorithms of the same class, identify potential parallelism, or apply known parallel patterns and optimization techniques. Furthermore, we also see an algorithm

classification as a way to facilitate communication among programmers, presenting them with a common idiom in which they can describe their computational problems. In these terms, our work shares many common goals with the work on pattern languages (e.g., Mattson et al. [2004]).

Second, we see an algorithm classification as a way to facilitate the design and improve the quality of tools and compilers such as auto-parallelizing compilers, source-to-source compilers, and auto-tuning compilers. Compilers will be able to base their parallelization and code transformation decisions on information embedded in the algorithm classes. This will greatly lift the burden of designing compilers, as they will not need to analyse the code to search for this information. Furthermore, algorithm classes might contain more information about the algorithm than is currently available to a compiler, giving opportunities to increase code quality. We envisage an algorithm classification to become a common compiler design technique, by bringing the extraction of algorithm details into a single, common place.

Having described the goals of an algorithm classification, we present a number of requirements that need to be met to achieve these goals.

- (1) Algorithm classes should be extracted automatically from program code. If they are not automatically extracted, the compilation process will involve manual work. Ultimately, this is not desirable, since the identification of classes can be error prone and places a heavy burden upon programmers.
- (2) A classification must be intuitive and easy to understand. Although we require classes to be extracted automatically, we also envisage a classification to be used manually, as a tool for programmers. In that case, an easy to learn and descriptive algorithm classification is paramount. Furthermore, an intuitive classification allows programmers to fine-tune the automatically extracted classes with information unavailable in the program code (e.g., knowledge about input values).
- (3) We require algorithm classes to be formally defined, i.e. code belongs to a certain class if and only if a set of formal properties hold. Such a definition helps guarantee correctness in compilers, allows programmers to fully understand properties of classes, and facilitates automatic extraction of classes from program code.
- (4) An algorithm classification must be complete within set boundaries. This means that a single algorithm should belong to at least one predefined class (possibly more than one if the classes overlap).
- (5) Classes must be fine-grained, capturing low-level algorithm details. Although a finer granularity is preferable, it must not come at the cost of other requirements. We require classes to include at least sufficient detail to be able to produce efficient code, i.e. the structure and amount of parallelism, and information on data reuse, providing opportunities to explore locality and caching.

With the goals and requirements for an algorithm classification set, we will look at existing algorithm classifications (although they might have different goals) and discuss whether they meet our requirements.

3. RELATED WORK

Algorithm classifications have been designed in the past for many different purposes, which has resulted in a large body of existing work with many different properties. We group this work in four categories, ranging from high abstraction-level classifications (loosely coupled to program code) to low abstraction-level classifications (tightly coupled to program code). We discuss the most prominent work of each category in the next sections.

```

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

```

Listing 1. Matrix-vector multiplication ($\vec{r} = \mathbf{M} \cdot \vec{v}$).

```

1 for (i=1; i<N-1; i++) {
2   for (j=1; j<N-1; j++) {
3     M[i][j] = 0.2 * (A[i][j]
4                   + A[i-1][j] + A[i][j+1]
5                   + A[i+1][j] + A[i][j-1]);
6   }
7 }

```

Listing 2. A 2D Jacobi stencil computation.

We identify the following categories, listed from high to low abstraction level:

- (1) high abstraction-level classifications, such as the Berkeley *motifs* [Asanovic et al. 2009], pattern languages [Keutzer and Mattson 2010; Massingill et al. 1999; Mattson et al. 2004], and the Galois system [Pingali et al. 2011];
- (2) algorithmic skeletons [Cole 1991] and related classifications, such as classical skeletons [Campbell 1996], contemporary skeletons [Caarls et al. 2006; Enmyren and Kessler 2010; Nugteren and Corporaal 2011; 2012a], and *idioms* [Carrington et al. 2011];
- (3) directive-based classifications, tightly coupled to program code. The OpenACC directives are an example;
- (4) mathematical representations of code, such as *Æcute* [Howes et al. 2009], the polyhedral model [Feautrier 1991; Pouchet et al. 2007], and the SUIF loop transformation formulation [Wolf and Lam 1991].

We discuss these classifications in more detail in the following sections, evaluating our goals and requirements. We illustrate a number of classifications by evaluating two example algorithms: matrix-vector multiplication (listing 1) and a 2D Jacobi stencil computation (listing 2). We provide a summary of our findings in Table I (Section 3.5).

3.1. High Abstraction-Level Classifications

The 13 *motifs*¹ (originally named *dwarfs*) from Berkeley [Asanovic et al. 2009] are an example of high abstraction-level classifications. Motifs are introduced as algorithmic methods to capture patterns of computation and communication. If we classify the examples, we find that the matrix-vector multiplication example of listing 1 fits nicely into the “dense linear algebra” motif, which is characterized by computations on dense matrices or vectors. The stencil computation of listing 2 is classified under the “structured grids” motif. Motifs are intended to be used for manual, high abstraction-level classification, yielding a coarse-grained intuitive classification, but lacking any automated extraction, formal definition, or completeness guarantee.

Related to motifs is the work on pattern languages for parallelism [Keutzer and Mattson 2010; Massingill et al. 1999; Mattson et al. 2004]. Pattern languages are intended to guide programmers by providing descriptions of frequently occurring problems. They typically provide patterns at multiple levels, but often start at a high abstraction level. An example is the pattern language OPL [Keutzer and Mattson 2010; Mattson et al. 2004], which uses motifs (named *computational patterns* in OPL) as a first classification step. A second step involves *structural patterns*, which describe the interaction of computational patterns. If we assume the output matrix $\mathbf{M}$ from the stencil computation (listing 2) to be used as input to the matrix-vector multiplication (listing 1), we can classify the sequence of examples as a “pipe-and-filter” structural pattern. The pattern language furthermore provides patterns for parallel programming. For both our examples we can select the “data parallelism” *algorithm strategy*, the “loop parallelism”

¹We use *italics* when introducing classification names and “quotes” for class names.

implementation strategy, and the “SIMD” *parallel execution pattern*. Although more detailed than motifs, OPL is still intended for manual classification purposes, making it unsuitable to meet both our goals.

The Galois system [Pingali et al. 2011] provides another high abstraction-level classification for manual uses. In contrast to other classifications, Galois is focused entirely on the classification of irregular algorithms. This makes the Galois system orthogonal to our work.

3.2. Algorithmic Skeletons and Related Classifications

Work on algorithmic skeletons [Cole 1991] has led to a large number of algorithm classifications. A summary of *classical* skeletons is found in a skeleton survey [Campbell 1996], listing skeletons such as farm, pipe, fork-join, divide-and-conquer, client-server, and zip. This survey of common algorithmic skeletons concludes with a general classification, capturing many skeletons from existing work. We use this classification to evaluate the examples. In the matrix-vector multiplication (listing 1), each computation $M[i][j] * v[j]$ results in a partial result of a single element of vector $\vec{r}$, requiring recombination. This fits the “recursively partitioned” or “divide-and-conquer” skeleton. The stencil computation (listing 2) computes a result directly, making it fit the “task queue” or “farm” skeleton. Such classical skeletons are very intuitive, but provide no automation, no completeness guarantees, no formal definition, and are too coarse-grained to meet our goals.

More recent *contemporary* skeleton work [Caarls et al. 2006; Enmyren and Kessler 2010; Nugteren and Corporaal 2011; 2012a] uses lower abstraction-level classifications. Example skeletons are map, reduce, map-reduce, map-overlap, and map-array [Enmyren and Kessler 2010] or pixel-to-pixel, neighborhood-to-pixel, pixel-to-global, and bucket processing [Caarls et al. 2006]. Related to the recent skeleton work are *idioms* [Carrington et al. 2011], a classification system defining 6 classes: stream, transpose, gather, scatter, reduction, and stencil. When classifying the examples using contemporary skeletons and idioms, we find the following results. The 2D Jacobi stencil computation of listing 2 classifies as “map-overlap” [Enmyren and Kessler 2010], “neighborhood-to-pixel” [Caarls et al. 2006], or as the equivalent “stencil” [Carrington et al. 2011]. However, these three classification techniques are unable to classify the full matrix-vector multiplication example, although the example can still be classified partially: the computation in the inner loop j can be classified as “reduce” [Enmyren and Kessler 2010], “scalar reduction” [Caarls et al. 2006] or “reduction” [Carrington et al. 2011]. Compared to classical skeletons, contemporary skeletons and idioms are already a better fit for our goals: they are formally defined in some cases (e.g., Caarls et al. [2006]), and occasionally provide tools for automation (e.g., Carrington et al. [2011]). Nevertheless, we cannot identify a single skeleton classification which fulfils all requirements, lacking aspects such as completeness and granularity for example.

3.3. Directive-Based Classifications

Compiler directives such as OpenHMPP and OpenACC² are tightly coupled to program code. Although directives are not strictly considered algorithm classifications, they do have the possibility to capture information on code segments. OpenACC for example is used by various compilers to specify regions of code to be offloaded to accelerators. It is used by for example PGI Accelerator [Wolfe 2010] and HMPP Workbench [Dolbeau et al. 2007]. As more OpenACC directives are added to program code, an increasing amount of information will become available to the compiler. For example, adding

²Details on OpenACC can be found at <http://www.openacc-standard.org/>.

“#pragma acc parallel loop” to the first lines of both examples (listings 1 and 2) will already set a specific “class”. We conclude that directives, although related to algorithm classifications, are missing a real notion of classes. Furthermore, they lack properties such as automated extraction.

3.4. Mathematical Code Representations

The final group of classifications do not introduce algorithm classes as such, but rather give a mathematical representation of algorithm code. Such mathematical representations typically work on loop nests and represent aspects such as iteration spaces, reuse distances, loop dependences, and data locality in a mathematical formulation. Examples are *Æcute* [Howes et al. 2009], the *polyhedral model* [Feautrier 1991; Pouchet et al. 2007], and the *SUIF loop transformation formulation* [Wolf and Lam 1991]. These mathematical formulations or models are often used for loop transformations such as tiling, skewing, and fusion.

Æcute, a decoupled access/execute specification [Howes et al. 2009], is an algorithm classification that uses a mathematical formulation. It has been used successfully in several works (e.g., Membarth et al. [2011]). The specification contains a description of the iteration space I , a precedence relationship R to set order of execution, a partition P to indicate sets of iterations preferably executed on a single processing element, and a set of memory locations that may be read (M_r) or written (M_w) for a given iteration. The *Æcute* specification for the two examples in listings 1 and 2 are shown in Equations (1) and (2) respectively. For similar examples and a detailed explanation we refer to Howes et al. [2009]. Although *Æcute* provides a fine-grained, complete, and formally defined classification, it is currently not automated, and less intuitive compared to classifications of higher abstraction level. Furthermore, the formulation of the partition requires knowledge of the target platform, making its identification less straightforward and, more importantly, not portable across different target platforms.

$$\begin{aligned}
 I &= \{(i, j) : 0 \leq i < N, 0 \leq j < N\} & M_r(i, j) &= \{M[i][j], v[j] : (i, j) \in I\} \\
 R &= \{((i, j), (i, k)) : 0 \leq i < N, 0 \leq j < k < N\} & M_w(i, j) &= \{r[i] : (i) \in I\} \\
 P &= \left\{ \{(i, j) \in I : a(k-1) \leq i < ak, b(l-1) \leq j < bl\} : 1 \leq k < \frac{N}{a}, 1 \leq l < \frac{N}{b} \right\}
 \end{aligned} \tag{1}$$

$$\begin{aligned}
 I &= \{(i, j) : 1 \leq i < N-1, 1 \leq j < N-1\} & M_w(i, j) &= \{M[i][j] : (i, j) \in I\} \\
 R &= \emptyset & M_r(i, j) &= \{A[i+x][j+y] : (i, j) \in I, -1 \leq x, y \leq 1\} \\
 P &= \left\{ \{(i, j) \in I : a(x-1) \leq i-1 < ax, b(y-1) \leq j-1 < by\} : \right. \\
 &\quad \left. 1 \leq x < \frac{N-2}{a}, 1 \leq y < \frac{N-2}{b} \right\}
 \end{aligned} \tag{2}$$

The polyhedral model captures the iteration space, the array references, and the iteration ordering of affine loop nests in a mathematical model. For our examples, we discuss only the iteration space in the domain description $\mathcal{D}$ and the array references in the access function f . For brevity, we show only the domain description for the innerloops and give f only for the access $M[i][j]$ in line 4 of listing 1 and for $A[i][j-1]$ in line 5 of listing 2. Detailed descriptions for these examples can be found in Section 4.2 and in Pouchet et al. [2007]. The resulting expressions are Equations (3) (matrix-vector

multiplication) and (4) (stencil computation).

$$\mathcal{D} = \{(i, j) | 0 \leq i, j < N\} \quad f(i, j) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \quad (3)$$

$$\mathcal{D} = \{(i, j) | 0 \leq i, j < N\} \quad f(i, j) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j-1 \end{pmatrix} \quad (4)$$

Although not strictly an algorithm classification, we do observe that the polyhedral model satisfies many of our requirements. It provides automatic extraction from source code, guarantees completeness within predefined bounds, obtains the finest granularity possible (it captures all information from the original code), and has a formal base. However, due to the nature of the polyhedral model (or other code representations for that matter), there is no real notion of classes, making it unsuitable to fulfil our goals.

3.5. Evaluation of Existing Classifications

We evaluate the results of the classification of the two examples for a number of different classifications³. We do this by assessing the quality of the classifications with respect to the defined requirements: we require algorithm classes to be: (1) automatically extracted, (2) intuitive, (3) formally defined, (4) complete, and (5), fine-grained. In Table I, we give an overview of the results of the classification of the two examples in the second column, and evaluate the requirements in the remaining columns. We discuss the evaluation of requirements for the existing classifications one by one.

- (1) We discussed two classifications that do provide automatic extraction of classes from program code. We identify an idiom recognizer [Carrington et al. 2011] and several polyhedral model extraction tools, e.g. PET [Verdoolaege and Grosser 2012]. The reasons that many classifications do not provide such tools can be found in the fact that the classes are either not formally defined or are too abstract, making the design of such a tool challenging or even impossible.
- (2) A large number of classifications, including motifs, pattern languages, and skeletons, use descriptive vocabulary as class names. This makes the classification intuitive and easy to understand for manual uses. In contrast, the mathematical descriptions provided by *Æcute* and the polyhedral model are less suited to fulfil goals for which higher abstraction-level classifications (such as motifs, pattern languages, and skeleton classifications) were introduced. The polyhedral model, due to the lack of a notion of classes, will for example not be directly applicable to tasks requiring an intuitive and compact class description, e.g. manual classification, guiding of programmers, code pattern identification, or skeleton selection.
- (3) A single skeleton classification, *Æcute*, and the polyhedral model provide formal definitions of classes. The other classifications rely on textual descriptions and examples, which might lead to errors, ambiguity, and lack of clarity.
- (4) Classifications such as pattern languages and algorithmic skeletons are often not complete. They can be extended with new classes as they evolve, but might still be unable to classify certain algorithms. In contrast, classifications using a mathematical description are complete, at least within predefined boundaries. For example,

³Although *Æcute* and the polyhedral model are not strictly classifications with individual classes, we do use these notations here to improve readability.

Table 1. Classification Findings for Two Example Algorithms Including an Evaluation of Our Requirements

	Classification	Auto- matic	Intuitive	Formally defined	Com- plete	Granu- larity
Motifs	ex. 1: Dense linear algebra ex. 2: Structured grids	no	yes	no	no	very coarse
OPL	ex. 1: Dense linear algebra + data par. + loop par + SIMD ex. 2: Structured grids + data par. + loop par + SIMD	no	yes	no	no	coarse
Skeletons	ex. 1: Recursive partitioned or di- vide & conquer ex. 2: Task queue or farm	no	yes	no	no	coarse
Enmyren et al.	ex. 1: Partial: reduce ex. 2: Map-overlap	no	yes	no	no	average
Caarls et al.	ex. 1: Partial: scalar reduction ex. 2: Neighbourhood to pixel	no	yes	yes	no	average
Idioms	ex. 1: Partial: reduction ex. 2: Stencil	yes	yes	no	no	average
Æcute	ex. 1: See equation 1 ex. 2: See equation 2	no	no	yes	yes	very fine
Polyhedral model	ex. 1: See equation 3 ex. 2: See equation 4	yes	no	yes	yes	very fine

the polyhedral model will be able to generate a domain description, access description, and iteration schedule for every affine loop nest.

- (5) The granularity of classifications increases gradually as they become less abstract and closer to program code. On one end are the motifs, grouping a wide range of algorithms into 13 abstract classes. On the other end are mathematical loop representations, describing data accesses and loop iterations in detail.

Concluding the evaluation, we can state that we have seen a large spectrum of classification techniques, each with its own benefits. However, none of the discussed classifications can satisfy all our requirements. Mathematical representations of program code such as Æcute and the polyhedral model come close, but lack a real notion of classes and thus fall short on intuitiveness, compactness, and ease of understanding. Such classifications are not suitable to meet our goals for manual uses. Other classifications such as skeletons or idioms do provide a descriptive classification, but lack aspects such as automation, granularity, and completeness. Therefore, we believe that a new algorithm classification is required to fulfil our requirements and to meet our goals.

4. ALGORITHMIC SPECIES

In the previous section we have shown that no existing algorithm classification fulfils our requirements. We therefore introduce a new classification in this section: algorithmic species. Algorithmic species⁴ is a mathematically defined classification that builds upon the polyhedral model [Feautrier 1991; Pouchet et al. 2007]. The classification defines classes (or: “species”) at a low abstraction level, classifying individual or nested affine loops, i.e. loops with affine array accesses and static affine loop control. The classes themselves are inspired by earlier skeleton classifications [Caarls et al. 2006; Enmyren and Kessler 2010; Nugteren and Corporaal 2011].

⁴Algorithmic species (or “species” in short) is both used as the name of the classification as well as to describe individual classes. Note that “species” is both the plural and the singular form.


```

1 for (i=0; i<64; i++) {
2   for (j=0; j<128; j++) {
3     R[i][j] = 2 * M[i][j];
4   }
5 }
    
```

Listing 3. An embarrassingly parallel algorithm.

Fig. 1. Illustration of the first two iterations of listing 3.

```

1 for (i=0; i<64; i++) {
2   S: r[i] = 0;
3   for (j=0; j<128; j++) {
4     T: r[i] += M[i][j] * v[j];
5   }
6 }
    
```

Listing 4. Matrix-vector multiplication, similar to listing 1.

Fig. 2. Illustration of the first two iterations of listing 4.

```

1 for (i=1; i<128-1; i++) {
2   m[i] = 0.33 *
3     (a[i-1] + a[i] + a[i+1]);
4 }
    
```

Listing 5. A 1D version of the 2D Jacobi stencil computation from listing 2.

Fig. 3. Illustration of the first two iterations of listing 5.

4.1. Examples

Prior to introducing the theory behind algorithmic species, we introduce several species informally by classifying the code examples of listings 3–7. Table II gives an overview of the classes found for the examples, which are illustrated through Figures 1–5. In these figures, different colors denote different iterations of the i -loop. The theory and further details are introduced in Section 4.3.

In listing 3 we give an example of a loop nest for which all iterations can be executed independently of each other. In the body of this example a single *element* from array $M[] []$ is read. After multiplying by 2, a single resulting *element* of array $R[] []$ is produced. Both arrays are accessed from indices 0 to 63 in the first dimension and from 0 to 127 in the second dimension. If we combine this data with the names of the arrays, we obtain the result as shown in Table II. The result can be interpreted as: on every iteration of the first dimension (0 to 63) and second dimension (0 to 127) one element is needed from the input array $M[] []$ to produce one element of the output array $R[] []$.

A second example covers the matrix-vector multiplication code found in listing 4 (the same as listing 1). Here, production of a single element of $r[]$ requires an entire row of array $M[] []$ and the complete array $v[]$. We identify these accesses as: *chunk* for the row access of $M[] []$ and *full* for the complete access of $v[]$. The resulting algorithmic

```

1 for (i=0; i<8; i++) {
2   r[0] += a[i] + b[i+2];
3 }

```

Listing 6. An example of reduction to a scalar.

```

1 for (i=0; i<2; i++) {
2   for (j=0; j<2; j++) {
3     R[i][j] = M[2*i][2*j] +
4               M[2*i+1][2*j] +
5               M[2*i][2*j+1] +
6               M[2*i+1][2*j+1];
7   }
8 }

```

Listing 7. An example of a 2D *chunk* access.

Fig. 4. Illustration of all 8 iterations of listing 6.

Fig. 5. Illustration of all 4 iterations of listing 7.

Table II. Classification of the Examples in Listings 3–7 as Algorithmic Species

Example	Algorithmic species
Listing 3	$M[0:63,0:127] \text{element} \rightarrow R[0:63,0:127] \text{element}$
Listing 4	$M[0:63,0:127] \text{chunk}(-,0:127) \wedge v[0:127] \text{full} \rightarrow r[0:63] \text{element}$
Listing 5	$a[1:126] \text{neighbourhood}(-1:1) \rightarrow m[1:126] \text{element}$
Listing 6	$a[0:7] \text{element} \wedge b[2:9] \text{element} \rightarrow r[0:0] \text{shared}$
Listing 7	$M[0:3,0:3] \text{chunk}(0:1,0:1) \rightarrow R[0:1,0:1] \text{element}$

species is shown in Table II, which can be interpreted as: to produce a single element out of the total 64 elements in $r[]$, the entire array $v[]$ of size 128 and a chunk of data in the second dimension of $M[]$ are needed.

Next, we look at the 1D stencil computation given in listing 5 (a 1D version of the 2D Jacobi stencil of listing 2). To produce a single *element* of array $m[]$, a *neighborhood* of 3 elements from $a[]$ is needed. A *chunk* access and a *neighborhood* access differ from each other in the fact that the latter implies overlap between subsequent iterations, as is the fact in the stencil example. The full classification can be found in Table II, in which the size of the neighborhood is given (ranging from -1 to $+1$).

In the reduction example of listing 6, we can see that for every input *element* of arrays $a[]$ and $b[]$ a contribution to the result $r[]$ is made. Since this is only a partial contribution, the result is considered *shared*. The classification that captures this behavior can be found in Table II. This captures the offset access to array $b[]$ by specifying the ranges from 2 up to and including 9.

Finally, we classify the example of listing 7. In this example, we require a 2x2 tile from $M[]$ to produce a single output in $R[]$. The tile is classified as a two-dimensional *chunk* access, resulting in the classification as shown in Table II.

4.2. The Polyhedral Model

The polyhedral model [Feautrier 1991; Pouchet et al. 2007] captures the iteration space, the array references, and the iteration ordering of loop nests in a mathematical model. The model is often used to perform loop transformations such as tiling, skewing, re-ordering, and fusion. As basis for algorithmic species, we use the polyhedral's *domain descriptions*, capturing the iteration space, and its *access functions*, capturing the array references. Although other mathematical models or dependence analyses might also be suitable for our purpose, we choose the polyhedral model for its formal representation, scalability, and for the availability of tools to extract a polyhedral representation from program code (e.g., Verdoolaege and Grosser [2012]). We illustrate the domain descrip-

tions $\mathcal{D}$ and the access functions f as used in the polyhedral model with the example matrix-vector multiplication code found in listing 4.

In the matrix-vector multiplication example (listing 4) we identify two statements: S in line 2 and T in line 4. For the statement S we describe the domain $\mathcal{D}_S$ as $\{i | 0 \leq i \leq 63\}$, which we can write in homogeneous coordinates as shown in Equation (5). The array access to array r in this statement is expressed as $r[f(\vec{t}_S)]$, for which $\vec{t}_S$ is a vector⁵ containing the loop iterators at statement S , any loop bound variables, and the constant 1. The access function $f(\vec{t}_S)$ for this access is given in Equation (6).

$$\mathcal{D}_S = \mathbf{D}_S \cdot \vec{t}_S = \begin{bmatrix} 1 & 0 \\ -1 & 63 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} \geq \vec{0} \quad (5)$$

$$f(\vec{t}_S) = \mathbf{F}_{S,r} \cdot \vec{t}_S = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} = (i) \quad (6)$$

Similarly to statement S we can derive the domain description $\mathcal{D}_T$ for statement T (see Equation (7) and the access to array $M[f(\vec{t}_T)]$ (see Equation (8)). For the access to array M we obtain a vector of length 2, which corresponds to the two-dimensional access to array M in statement T . For brevity, we omit the access functions for arrays r and v in statement T .

$$\mathcal{D}_T = \{(i, j) | 0 \leq i \leq 63 \wedge 0 \leq j \leq 127\} = \mathbf{D}_T \cdot \vec{t}_T = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 63 \\ 0 & 1 & 0 \\ 0 & -1 & 127 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \vec{0} \quad (7)$$

$$f(\vec{t}_T) = \mathbf{F}_{T,M} \cdot \vec{t}_T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \quad (8)$$

Additionally, we represent the accesses to another array M in a second example (listing 3) using the polyhedral model. If we derive the domain description and the access function for the access to M in this example, we observe that they are equal to those found for the matrix-vector multiplication example (Equations (7) and (8)). In other words, because the access $M[i][j]$ uses the same indices in both listings 3 and 4, the polyhedral model describes both accesses equally.

If we evaluate these results with respect to the classifications in Table II for these examples, we see that the accesses to array M are classified distinctly, either as *element* (listing 3) or as *chunk* (listing 4). The reason for a distinguished classification can be found in the fact that accesses to array r in the matrix-vector multiplication of listing 4 impose data dependences, limiting the parallelism in M . The polyhedral model is unable to expose this directly, because its access functions describe individual array accesses in isolation of others. To be able to distinguish between the array accesses to M in these examples, we introduce a new representation of array accesses along with the theory of algorithmic species.

⁵Although the vector $\vec{t}$ is typically named $\vec{x}$ in polyhedral representations, we use this notation to prevent confusion with a vector $\vec{x}$ we introduce in Section 4.3. Similarly, we use $\mathbf{D}_S$ rather than $\mathbf{A}_S$ for the domain descriptions.

4.3. Deriving Algorithmic Species

In this section we introduce algorithmic species formally. Algorithmic species are constructed from a combination of “array access patterns”⁶. We define five such patterns, which are already introduced by the example species found in Table II: *element*, *chunk*, *neighborhood*, *shared*, and *full*. These access patterns are specific for a single array access in a statement, while an algorithmic species covers a complete loop or loop nest. Key to the algorithmic species approach is the use of these access patterns as building blocks, which enables the creation of an unlimited amount of different species using only a limited set of access patterns.

In Section 4.2 we argued that access functions in the polyhedral model do not directly lead to access patterns. Therefore, before defining the access patterns formally, we define a new notation for access functions and domain descriptions by distinguishing two types of loops.

4.3.1. Base Loops and Structure Loops. We identify two types of loops by splitting up the vector $\vec{t}$ as used in the polyhedral model. We name the two types *base loops* ($\vec{x}$) and *structure loops* ($\vec{y}$). Formally, we define a structure loop as a loop containing either: (1) a read access that is dependent on the loop iterator while all write accesses are not, or (2), a write access that is dependent on the loop iterator while all read accesses are not. All other loops which contain at least an array access are considered base loops. Structure loops can be seen as loops distorting a one-to-one access-to-iteration mapping of arrays in a base loop, creating accesses of for example rows, columns, tiles, and neighborhoods. An example is the j -loop in the matrix-vector multiplication example (listing 4). A procedure to derive the loop type from code is given in Algorithm 1.

We illustrate Algorithm 1 by evaluating the matrix-vector multiplication example of listing 4. First, we construct the modified vectors $\vec{t}'$ by removing any constants from $\vec{t}$ and update the access matrices $\mathbf{F}$ accordingly, creating $\mathbf{F}'$. The results for statements S and T of the matrix-vector multiplication (listing 4) are shown in Equations (9) (for S) and (10) (for T).

$$\vec{t}_S = \begin{pmatrix} i \\ 1 \end{pmatrix} \quad \rightarrow \quad \vec{t}'_S = (i), \quad \mathbf{F}'_{S,r} = [1] \quad (9)$$

$$\vec{t}_T = \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \quad \rightarrow \quad \vec{t}'_T = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \mathbf{F}'_{T,r} = [1 \ 0], \quad \mathbf{F}'_{T,M} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{F}'_{T,v} = [0 \ 1] \quad (10)$$

Since statement S does not have any read accesses, we cannot derive any structure loops from S . As for statement T , we can continue Algorithm 1 and construct the vectors $\vec{R}_T$ and $\vec{W}_T$. They are constructed by summing all values per column (projection) of all $\mathbf{F}'_{T,a}$ matrices for which a is a read or a write access, respectively. The results are shown in Equation (11).

$$\vec{R}_T = [1 \ 1] + [0 \ 1] = [1 \ 2] \quad \vec{W}_T = [1 \ 0] \quad (11)$$

If we look at the loop iterator i , and thus at the first column of $\vec{R}_T$ and $\vec{W}_T$, we see no values equal to zero ($R_{T,0} = W_{T,0} = 1$). However, when looking at the second loop iterator j , we can find a read which is dependent of the loop iterator ($R_{T,1} \neq 0$) and find that all writes are independent ($W_{T,1} = 0$). Thus, we identify j as a structure loop. Therefore, we find $\vec{x} = (i)$ and $\vec{y} = (j)$ as result of Algorithm 1.

⁶Access patterns used for algorithmic species have no relation with pattern languages.

ALGORITHM 1: Algorithm to deriving base loops and structure loops for a loop body.

Input: Polyhedral description of all statements S in the loop body

Output: Base loop vector $\vec{x}$ and structure loop vector $\vec{y}$
 $\vec{x} = \emptyset; \vec{y} = \emptyset$
for each statement S in S do
 $\vec{t}'_S \leftarrow \vec{t}_S$ without constants

 $n = \text{length}(\vec{t}'_S)$
for all accesses a in S do
 $\mathbf{F}'_{S,a} \leftarrow$ the first n columns of $\mathbf{F}_{S,a}$
end
 $\vec{R}_S \leftarrow$ projection of $\mathbf{F}'_{S,a}$ into a row vector, for which a are read accesses

 $\vec{W}_S \leftarrow$ projection of $\mathbf{F}'_{S,a}$ into a row vector, for which a are write accesses

if $\vec{R}_S = \vec{0}$ or $\vec{W}_S = \vec{0}$ then
 $\text{continue};$
end
for i in $[0 : n - 1]$ do
if $(R_{S,i} \neq 0 \text{ and } W_{S,i} = 0) \text{ or } (R_{S,i} = 0 \text{ and } W_{S,i} \neq 0)$ then
 $\vec{y} \leftarrow \vec{y} \cup \vec{t}_{S,i}$
end
end
end
 $\vec{x} \leftarrow \vec{t}_S \cap \vec{y}$
Result: $\vec{x}, \vec{y}$

4.3.2. Domain Descriptions and Access Functions. We introduce a new way to describe iteration domains and access functions based on the splitting of loops into base loops and structure loops. In Section 4.2, we showed that for the polyhedral model the domain description $\mathcal{D}$ and the access function $f(\vec{t})$ are given in the form of $\mathbf{D} \cdot \vec{t}$ and $\mathbf{F} \cdot \vec{t}$ respectively. Since the vector $\vec{t}$ is split into the loop vectors $\vec{x}$ and $\vec{y}$ and into a separate constant vector, we require alternate domain descriptions and access functions.

Because the loop iterators are now split into two vectors ($\vec{x}$ and $\vec{y}$), we also split the domain description in two parts: one part describes the domain imposed by the base loops ($\mathcal{D}_x$) and another part describes the domain imposed by the structure loops ($\mathcal{D}_y$). To illustrate this, we take statement T of the matrix-vector multiplication example (listing 4) as an example. We split Equation (7), the domain description in the polyhedral model, into two new descriptions, which are shown in Equation (12).

$$\mathcal{D}_{x,T} = \begin{bmatrix} 1 & 0 \\ -1 & 63 \end{bmatrix} \cdot \begin{pmatrix} i \\ 1 \end{pmatrix} \geq \vec{0} \quad \mathcal{D}_{y,T} = \begin{bmatrix} 1 & 0 \\ -1 & 127 \end{bmatrix} \cdot \begin{pmatrix} j \\ 1 \end{pmatrix} \geq \vec{0} \quad (12)$$

The access functions to represent array accesses are changed as well. We introduce two new matrices: $\mathbf{A}$ gives the relation between the array indices and the base loop iterators $\vec{x}$, while $\mathbf{B}$ gives the relation between the array indices and the structure loop iterators $\vec{y}$. Additionally, we introduce $\vec{c}$ to set a constant offset. Together, this creates an access function $\vec{I}$, written as $\vec{I}_a$ for array access a . The new access function can be found in Equation (13). Note that in this access function the loop iterators $\vec{x}$ and $\vec{y}$ are set for an entire loop body instead of for an individual statement as is done for $\vec{t}$ in the polyhedral model.

$$\vec{I}_a = \mathbf{A}_a \cdot \vec{x} + \mathbf{B}_a \cdot \vec{y} + \vec{c}_a \quad (13)$$

We give a number of example access functions for statement T of the matrix-vector multiplication (listing 4) for the accesses to arrays r (Equation (14)), m (Equation (15)),

and v (Equation (16)). In this example, we can quickly see that the accesses to arrays r and v are one-dimensional (a single row in Equations (14) and (16), while the access to array M is two-dimensional (two rows in Equation (15)).

$$\mathbf{A}_r = [1] \quad \mathbf{B}_r = [0] \quad \vec{c}_r = [0] \quad \vec{I}_r = [1] \cdot \vec{x} + [0] \cdot \vec{y} + [0] = (i) \quad (14)$$

$$\mathbf{A}_M = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \mathbf{B}_M = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \vec{c}_M = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{I}_M = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \vec{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot \vec{y} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \quad (15)$$

$$\mathbf{A}_v = [0] \quad \mathbf{B}_v = [1] \quad \vec{c}_v = [0] \quad \vec{I}_v = [0] \cdot \vec{x} + [1] \cdot \vec{y} + [0] = (j) \quad (16)$$

With the new access function set, we re-evaluate the problem of distinguishing the array accesses of the matrices M in listings 3 and 4. Although the indices are the same for both examples ($M[i][j]$) and the polyhedral model gives the same description, the new access function does give a distinguished description. The access function for array M in the matrix-vector multiplication (listing 4) is already given in Equation (15), while the access function for array M in listing 3 is given in Equation (17). The difference is that $\vec{x}$ contains both loop iterators for listing 3, but only the outer loop iterator for listing 4.

$$\mathbf{A}_M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{B}_M = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{c}_M = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{I}_M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \vec{x} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \cdot \vec{y} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \quad (17)$$

Now, when we compare the descriptions for array M in Equation (15) and (17), we find different matrices $\mathbf{A}$ and $\mathbf{B}$ and can thus distinguish between the *element* and *chunk* patterns. In the next section, we show how the matrices $\mathbf{A}$ and $\mathbf{B}$ are used to derive each of the five access patterns.

4.3.3. Deriving the Access Patterns. Based on the domain descriptions and access function as introduced, we can derive access patterns. The total of five access patterns are given descriptive names to create an intuitive and easy to understand classification. The patterns, which were already illustrated by the examples found in listings 3–7, are defined formally in Algorithm 2. This algorithm defines patterns based on the matrices $\mathbf{A}$ and $\mathbf{B}$ and the domain descriptions ($\mathcal{D}_x, \mathcal{D}_y$) for a given array access in a given statement. In words, Algorithm 2 defines the *element* pattern as accesses only dependent on base loops, the *full* pattern as accesses only dependent on structure loops, the *chunk* and *neighborhood* patterns as accesses dependent on both types of loops, and the *shared* pattern as accesses independent of both base loops and structure loops. The *neighborhood* pattern is distinguished from the *chunk* pattern by the fact that the pattern implies partial overlap among iterations. Partial overlap exists if Equation (18) holds.

$$\begin{aligned} &\exists(\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2) \text{ such that } \mathbf{A}_a \cdot \vec{x}_1 + \mathbf{B}_a \cdot \vec{y}_1 = \mathbf{A}_a \cdot \vec{x}_2 + \mathbf{B}_a \cdot \vec{y}_2, \\ &\text{with } (\vec{x}_1 \neq \vec{x}_2 \vee \vec{y}_1 \neq \vec{y}_2) \text{ and } (\vec{x}_1, \vec{x}_2) \in \mathcal{D}_{x,S} \text{ and } (\vec{y}_1, \vec{y}_2) \in \mathcal{D}_{y,S} \end{aligned} \quad (18)$$

Limitations of parallelism in the form of dependences are not captured by Algorithm 2. Therefore, we further require arrays to have no dependences with themselves. Arrays can depend on themselves through a chain of zero (e.g., $A[i+1] = A[i]$) or more statements in the same loop or in multiple loops. As is illustrated in Allen and Kennedy [1987], when no array depends on itself, the loop (nest) can be vectorized or parallelized. The theory as in Allen and Kennedy [1987] can be applied to the algorithmic species in

ALGORITHM 2: Algorithm to derive the access patterns for an array.

Input: Access description matrices ($\mathbf{A}_a$, $\mathbf{B}_a$) and domain descriptions ($\mathcal{D}_{x,S}$, $\mathcal{D}_{y,S}$) for array access a in statement S

Output: Access pattern $\mathcal{P}_a$ for array access a

```

if ( $\mathbf{B}_a = 0$ ) then
  if ( $\mathbf{A}_a = 0$ ) then
     $\mathcal{P}_a \leftarrow \text{"shared"}$ 
  else
     $\mathcal{P}_a \leftarrow \text{"element"}$ 
  end
else
  if ( $\mathbf{A}_a = 0$ ) then
     $\mathcal{P}_a \leftarrow \text{"full"}$ 
  else if (equation 18 holds, i.e. there is partial overlap) then
     $\mathcal{P}_a \leftarrow \text{"neighbourhood"}$ 
  else
     $\mathcal{P}_a \leftarrow \text{"chunk"}$ 
  end
end
Result:  $\mathcal{P}_a$ 

```

terms of the matrices $\mathbf{A}$ and $\mathbf{B}$ and the vector $\vec{c}$. The outcome of Algorithm 2 is therefore only valid if there is no single array that is read ($\vec{I}_{a,r}$) and written ($\vec{I}_{a,w}$) or written twice at the same location in different loop iterations. The loop iterations for which the dependences are checked must be within the loop bounds ($\mathcal{D}_x$ and $\mathcal{D}_y$). There are no dependences when the condition in Equation (19) holds. This can be implemented efficiently using for example the I-test [Kong et al. 1991].

$$\nexists(\vec{I}_{a,w}(\vec{x}_1, \vec{y}_1) = \vec{I}_{a,w}(\vec{x}_2, \vec{y}_2)) \quad \text{and} \quad \nexists(\vec{I}_{a,r}(\vec{x}_1, \vec{y}_1) = \vec{I}_{a,w}(\vec{x}_2, \vec{y}_2)), \quad (19)$$

with $(\vec{x}_1 \neq \vec{x}_2 \vee \vec{y}_1 \neq \vec{y}_2) \quad \text{and} \quad (\vec{x}_1, \vec{x}_2) \in \mathcal{D}_{x,S} \quad \text{and} \quad (\vec{y}_1, \vec{y}_2) \in \mathcal{D}_{y,S}$

To ensure that the classification is complete, Algorithm 2 is designed such that every type of array access is always described by one of the five access patterns. This implies that a species can be derived for every loop nest for which all array accesses can be described by Equation (13) and Equation (19) holds. The access patterns however do not complete the algorithmic species, as the ranges of the array, chunk, and neighborhood accesses are not derived yet. The following section derives these ranges formally.

4.3.4. Deriving the Ranges of Array and Pattern Accesses. The domain descriptions and access functions enable us to derive the range for which an array is accessed. Next to that, we can also derive the range of a *chunk* or *neighborhood* access.

In order to calculate the range of a chunk or neighborhood access, we need information from the domain of the structure loop ($\mathcal{D}_y$), the structure loop access matrix ($\mathbf{B}$), and the constant offset vector ($\vec{c}$). We give the procedure to derive the structure range in Algorithm 3, which we illustrate by example in this section. To explain the range calculation step-by-step we will look at statement T of the matrix-vector multiplication example (listing 4). For this example, we will derive the range and thus shape of the *chunk* access (a row access in this case). The domain of the structure loop is given in Equation (20), with the loop bounds visible in the last column of $\mathbf{D}_{y,T}$. Because the domain description is based on the polyhedral model [Verdoolaege and Grosser 2012], we can always obtain the loop bounds of the structure loops directly by taking the last

ALGORITHM 3: Algorithm to derive a range for a *chunk* or *neighbourhood* pattern.

Input: Array access patterns (S_a), array descriptions ($\mathbf{B}_a, \vec{c}_a$), structure loops ($\vec{y}$), and the domain description ($\mathcal{D}_{y,S}$) for array a in statement S

Output: Ranges S of a chunk or neighbourhood access

$S = \emptyset$

$S_a \leftarrow$ access pattern of array a

if ($S_a \in \{\text{"chunk"}, \text{"neighbourhood"}\}$) **then**

$\vec{y} \leftarrow$ structure loops

$\mathcal{D}_{y,S} \leftarrow$ domain description for statement S

$(\mathbf{B}_a, \vec{c}_a) \leftarrow$ access description of array a

$\mathbf{H}_{inv} \leftarrow$ construct inverter matrix

$\vec{h} \leftarrow$ construct row selector based on $\vec{y}$

$\mathbf{C}_a \leftarrow$ construct offset matrix based on $\vec{c}_a$

$S = \mathbf{B}_a \cdot \vec{h} \cdot \mathbf{D}_y^\top \cdot \mathbf{H}_{inv} + \mathbf{C}_a$

end

Result: S

column of $\mathbf{D}_{y,T}$ (or as used later: the last row of $\mathbf{D}_{y,T}^\top$).

$$\mathcal{D}_{y,T} = \mathbf{D}_{y,T} \cdot \begin{pmatrix} j \\ 1 \end{pmatrix} \geq \vec{0} \quad \text{with} \quad \mathbf{D}_{y,T} = \begin{bmatrix} 1 & 0 \\ -1 & 127 \end{bmatrix} \quad (20)$$

The value of the lower bound in matrix $\mathbf{D}_y$ is always negated. To invert the lower bound we introduce a helper matrix $\mathbf{H}_{inv}$, set for this example as $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$. To be able to extract the bounds of the loop, we additionally introduce a helper vector $\vec{h}$ to select the last row, set for this example as $[0 \ 1]$. The bounds of the loop for the example are extracted from $\mathbf{D}_{y,T}$ as is shown in Equation (2).

$$\vec{h} \cdot \mathbf{D}_{y,T}^\top \cdot \mathbf{H}_{inv} = [0 \ 1] \cdot \begin{bmatrix} 1 & -1 \\ 0 & 127 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = [0 \ 127] \quad (21)$$

To complete the range calculation, we process the stride of the access between successive structure loop iterations ($\mathbf{B}$) and the access offset which shifts the range with a constant amount ($\vec{c}$). The range of the *chunk* access to array M for the example is computed in Equation (22). Similar as discussed before for $\vec{l}$, each row in the range corresponds to a dimension in the array access. The range description has always two columns: the first denoting the start and the second the end of the range. The range for *chunk* and *neighborhood* patterns is given with respect to the base loop iterators. As shown in Equation (22) for the example, array M is accessed as a *chunk* only in the second dimension (i.e., a row access), ranging from 0 up to and including 127.

$$\mathbf{B}_M \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^\top \cdot \mathbf{H}_{inv}) + (\vec{c}_M \ \vec{c}_M) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot [0 \ 127] + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 127 \end{bmatrix} \quad (22)$$

To complete the algorithmic species, we finally include the range of the array accesses. The range is computed analogue to the *chunk* or *neighborhood* ranges, with the only differences that both matrices ($\mathbf{A}$ and $\mathbf{B}$) and both domain descriptions ($\mathbf{D}_x$ and of $\mathbf{D}_y$) are used. For the arrays r , M and s in the example we find the ranges as shown in Equations (23), (24), and (25) respectively.

$$\mathbf{A}_r \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^\top \cdot \mathbf{H}_{inv}) + \mathbf{B}_r \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^\top \cdot \mathbf{H}_{inv}) + (\vec{c}_r \ \vec{c}_r) = [0 \ 63] \quad (23)$$

$$\mathbf{A}_M \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^\top \cdot \mathbf{H}_{inv}) + \mathbf{B}_M \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^\top \cdot \mathbf{H}_{inv}) + (\vec{c}_M \ \vec{c}_M) = \begin{bmatrix} 0 & 127 \\ 0 & 63 \end{bmatrix} \quad (24)$$

$$\mathbf{A}_s \cdot (\vec{h} \cdot \mathbf{D}_{x,T}^\top \cdot \mathbf{H}_{inv}) + \mathbf{B}_s \cdot (\vec{h} \cdot \mathbf{D}_{y,T}^\top \cdot \mathbf{H}_{inv}) + (\vec{c}_s \ \vec{c}_s) = [0 \ 127] \quad (25)$$

4.3.5. Algorithmic Species Anatomy. With the access patterns and ranges derived we can construct algorithmic species. Arrays a which are accessed using the *element*, *shared* or *full* patterns are constructed using the array name ($\mathcal{N}_a$), the access range ($\mathcal{R}_a$), and the access pattern ($\mathcal{P}_a$). For the access patterns *chunk* and *neighborhood*, the additional structure range ($\mathcal{S}_a$) is included. The syntax used for algorithmic species is illustrated in Equations (26) for both types.

$$\begin{aligned} \mathcal{N}_a[\mathcal{R}_a]|\mathcal{P}_a & \quad \text{for which } \mathcal{P}_a \in \{\text{element, shared, full}\} \\ \mathcal{N}_a[\mathcal{R}_a]|\mathcal{P}_a(\mathcal{S}_a) & \quad \text{for which } \mathcal{P}_a \in \{\text{chunk, neighbourhood}\} \end{aligned} \quad (26)$$

The ranges itself are constructed using commas (,) to separate dimensions and colons (:) to separate the start and end of a range. Furthermore, multiple array accesses are concatenated with a wedge ($\wedge$), while input and output accesses are separated with an arrow ($\rightarrow$).

Combining all the above, we can construct algorithmic species as shown for a number of examples in Table II. Furthermore, algorithmic species can classify loops at any level. For example, we can also consider the inner loop j of listing 4 only. In that case, we consider i as a constant value and classify the inner loop as

$$M[i:i,0:127]|\text{element} \wedge v[0:127]|\text{element} \rightarrow r[i:i]|\text{shared}.$$

4.3.6. Unordered Accesses. So far, we have not discussed the relation between accesses to different arrays in a species. For example, if we change the body of listing 3 from “ $R[i][j] = 2 * M[i][j]$ ” into “ $R[i][j] = 2 * M[63-i][127-j]$ ”, we still read and write a single *element* every iteration of the loops i and j , and access the same ranges of data as specified in Table II. In order to distinguish between the two, we introduce the keyword *unordered*, which is added to the algorithmic species in case there are two or more matrices $\mathbf{A}$ in a species which are not equal to each other (excluding zero matrices). The keyword *unordered*, applied only if Equation (27) does not hold, captures accesses which are amongst others strided, reversed, transposed, or scattered. If the keyword is not specified, a relation is assumed for which accesses are ordered, starting at the lower bound of the specified range.

$$\forall (\mathbf{A}_1 \neq 0, \mathbf{A}_2 \neq 0) \quad \text{it holds that} \quad \mathbf{A}_1 = \mathbf{A}_2 \quad (27)$$

4.4. Limitations and Remarks

A number of limitations similar to those of the polyhedral model are imposed upon the algorithmic species theory. First of all, for code to be represented using our formulations, it must contain affine array accesses and static affine loop control, i.e. loop bounds that can be expressed as a system of affine inequalities and do not change throughout the execution of the loop. Furthermore, arrays must be accessed with explicit addresses, e.g. pointer arithmetic is not supported.

We make a remark on the fact that all ranges in algorithmic species are upper bounds only. For example, if an *element* access ranges from 0 to 63, it might be possible that elements 10 and 40 are never accessed at all. Similarly, ranges for *chunk* and *neighborhood* are upper bounds as well: partial *chunks* or partial *neighborhoods* are still classified as *chunks* or *neighborhoods*.

Finally, we note that the *full* pattern can be seen as a special case of the *chunk* pattern. When the size of the structure range of the *chunk* is equal to the access

range, it is classified as *full* according to Algorithm 2. The inclusion of the *full* in the algorithmic species is therefore not required for completeness, but it does improve the compactness and readability of the class names.

4.5. Summary

To conclude the theory of algorithmic species, we give a summary of the steps that need to be taken to derive a species from a given affine loop nest: (1) construct the domain descriptions and access functions in the polyhedral model (Section 4.2); (2) derive the base loops and structure loops (Algorithm 1); (3) compute the matrices D_x , D_y , A , B , and c (Section 4.3.2); (4) apply the dependence check (Equation (19)); (5) derive the access patterns (Algorithm 2); and (6), compute all access ranges (Algorithm 3). Because these steps are all deterministic formal steps, a resulting species is formally defined to correspond to its program code and vice versa.

5. VALIDATION

In this section, we classify a benchmark suite using the presented algorithmic species, and discuss the results. We also briefly discuss *ASET*, a tool to extract species from program code, thus fulfilling our requirement of automatic extraction. To further demonstrate the use of algorithmic species, we identify two automatic uses: “BONES” [Nugteren and Corporaal 2012a] and the “boat hull model” [Nugteren and Corporaal 2012b]. The first use, *BONES*, is a skeleton-based source-to-source compiler that performs skeleton selection based on species information. The second use, the boat hull model, is a performance prediction technique based on the roofline model [Williams et al. 2009] that introduces multiple species-specific roofline models.

5.1. Algorithmic Species in Practice

To validate the use of algorithmic species, we manually classify a large number of algorithms. For this purpose, we have chosen the PolyBench/C 3.2 benchmark suite,⁷ which consists of 30 algorithms (157 loops in total) selected from 6 domains in scientific computing, ensuring a wide variety of algorithms. These algorithms contain static control parts with one or more loop nests with affine loop bounds and array accesses. An overview of the algorithms in PolyBench grouped by domain is given in Table III.

We have classified code in the PolyBench suite using algorithmic species. The last two columns of Table III give an overview in terms of amount of species identified. We show the amount of nonnested species in the third column. Additional nested classifications are shown in between brackets, i.e., classifications of a loop and its body which are already part of another species. From the total of 30 algorithms, we are able to find 28 algorithms with loop parallelism. All loops with parallelism in these 28 algorithms are identified by a total of 115 species (not all necessarily unique), of which 55 species are part of others (i.e., nested species). All other loops do not meet our dependence check as given in Equation (19). For these loops a thorough inspection was performed, but we were unable to find any loop parallelism in their current form. The 115 species found are not shown here, but can be found online⁸.

From the total of 115 species identified, we select a few cases for which it is not directly clear that the loops can be executed in parallel. First, we take a detailed look at the last part of the *reg_detect* benchmark, shown in listing 8. We find that all iterations of the *i*-loop (lines 2–4) can be executed in parallel, i.e. Equation (19) holds. The corresponding species is as follows: `path[j-1:j-1, j-1:MAXGRID-2] | element`

⁷PolyBench can be found at: <http://www.cse.ohio-state.edu/pouchet/software/polybench/>.

⁸PolyBench annotated with algorithmic species can be found at: <http://parse.ele.tue.nl/species/>.

Table III. Classification Results for the PolyBench Benchmark Suite

Benchmark	Summary of functionality	#Species	SLoC
Linear algebra kernels			
2mm	Two matrix multiplications	2 (+4)	16/16 (100%)
3mm	Three matrix multiplications	3 (+6)	24/24 (100%)
atax	Matrix transpose and vector multiplication	2 (+2)	12/12 (100%)
bicg	BiCG sub kernel of BiCGStab linear solver	2 (+2)	12/12 (100%)
cholesky	Cholesky decomposition kernel	2 (+1)	10/14 (71%)
doitgen	Multi-resolution analysis kernel	2 (+5)	17/17 (100%)
gemm	Matrix multiplication	1 (+2)	8/8 (100%)
gemver	Matrix vector multiplication and matrix addition	4 (+3)	18/18 (100%)
gesummv	Scalar, vector and matrix multiplication	1 (+1)	9/9 (100%)
mvt	Matrix vector product and transpose kernel	2 (+2)	10/10 (100%)
symm	Symmetric matrix multiplication	1 (+2)	11/13 (85%)
syr2k	Symmetric rank-2k operations	1 (+2)	9/9 (100%)
syrk	Symmetric rank-k operations	1 (+2)	8/8 (100%)
trisolv	Solver for linear systems with triangular matrices	1	4/7 (57%)
trmm	Triangular matrix multiplication kernel	1	3/7 (43%)
Linear algebra solvers			
gramschmidt	Algorithm for the Gram-Schmidt process	3 (+2)	16/21 (76%)
durbin	Equation solver using the Levinson-Durbin recursion	2	6/18 (33%)
dynprog	2D dynamic programming algorithm	1 (+1)	5/17 (29%)
lu	LU decomposition kernel	2 (+1)	8/10 (80%)
ludcmp	LU decomposition kernel	4	12/33 (36%)
Data-mining			
correlation	Correlation computation kernel	4 (+4)	30/34 (88%)
covariance	Covariance computation kernel	3 (+3)	19/21 (90%)
Graph algorithms			
floyd-warshall	Find shortest path in weighted graph	0	0/7 (0%)
Image processing			
reg_detect	2D Regularity detection algorithm	3 (+2)	14/27 (52%)
Stencil operations			
adi	Alternating direction implicit solver	3	9/30 (30%)
fdtd-2d	2D finite different time domain kernel	4 (+3)	20/20 (100%)
fdtd-2d-apml	2D FDTD using an anisotropic perfectly matched layer	1 (+3)	28/28 (100%)
jacobi-1d-imper	1D Jacobi stencil computation	2	6/8 (75%)
jacobi-2d-imper	2D Jacobi stencil computation	2 (+2)	12/12 (100%)
seidel-2d	2D Seidel in-place stencil operation	0	0/7 (0%)

The third column lists the number of nonnested species and between brackets the additional number of nested species. The last column gives the amount and percentage of classified source lines of code (SLoC), i.e. with parallelism.

```

1 for (j=1; j<=MAXGRID-1; j++) {
2   for (i=j; i<=MAXGRID-1; i++) {
3     path[j][i] = path[j-1][i-1] + mean[j][i];
4   }
5 }

```

Listing 8. Code snippet taken from the reg_detect benchmark.

Fig. 6. Accesses of path for listing 8 with j=2 and MAXGRID=5.

$\wedge$ mean[j:j, j:MAXGRID-1] element $\rightarrow$ path[j:j, j:MAXGRID-1] element. We visualize the execution of the i -loop in Figure 6, in which we denote read accesses to the array path[] with R and write accesses with W.

Second, we discuss the last part of the covariance benchmark, which is shown in listing 9. Applying the algorithmic species theory, we find that each iteration

```

1 for (j1=0; j1<M; j1++) {
2   for (j2=j1; j2<M; j2++) {
3     symmat[j1][j2] = 0.0;
4     for (i=0; i<N; i++) {
5       symmat[j1][j2] += data[i][j1] * data[i][j2];
6     }
7     symmat[j2][j1] = symmat[j1][j2];
8   }
9 }

```

Listing 9. Code snippet taken from the covariance benchmark.

Fig. 7. Accesses of symmat for listing 9 with $j1=2$ and $N=M=5$.

of the $j2$ -loop (lines 2–8) can be executed in parallel. This loop is classified as follows: “ $\text{data}[0:N-1, j1:j1] | \text{full} \wedge \text{data}[0:N-1, j1:M-1] | \text{chunk}(0:N-1, 0:0) \rightarrow \text{symmat}[j1:M-1, j1:j1] | \text{element} \wedge \text{symmat}[j1:j1, j1:M-1] | \text{element}$ ”. We see that the input array $\text{data}[] []$ is accessed in two ways: as $(i, j1)$ and as $(i, j2)$. This gives us an overlapping *full* and *chunk* access. The output is also written in two different statements, which is visualized in Figure 7. In this figure, A denotes write and read access, and W denotes write-only access. Execution of the $j2$ -loop in parallel will yield correct results, since every $\text{symmat}[] []$ element that is written only depends on reads from $\text{symmat}[] []$ elements either accessed in the same iteration or not accessed in this iteration of the $j1$ -loop at all.

5.2. Automatic Extraction of Algorithmic Species

To automatically identify algorithmic species from program code, we developed ASET⁹ (algorithmic species extraction tool). ASET builds upon PET [Verdoolaege and Grosser 2012], a tool that is able to extract polyhedral information from C program code. Based on this information, ASET applies Algorithms 1–3 and all corresponding equations as shown in Section 4.3, identifying algorithmic species according to the definitions. The tool annotates the original source code, delimiting the species with “`#pragma species kernel <class>`” and “`#pragma species endkernel`”. Applying ASET to PolyBench leads to the same results as the manual classification as presented in Table III.

5.3. Using Algorithmic Species in a Source-to-Source Compiler

We demonstrate the automatic use of algorithmic species by presenting the BONES¹⁰ source-to-source compiler [Nugteren and Corporaal 2012a]. The compiler is based on *algorithmic skeletons* [Cole 1991], a technique using parametrizable program code (*skeletons*) to generate high performance code. A single skeleton can be seen as templated code for a specific class of computations on a specific target processor. Skeletons can be added as new classes are identified, creating a flexible compiler. Typically, users of skeleton-based compilers are required to manually select a suitable skeleton for their algorithm. However, in the case of BONES, algorithmic species information is used to automatically select a skeleton for a given algorithm. This makes BONES, combined with an automatic species extraction tool such as ASET, a fully automatic source-to-source compiler. Details on BONES are available in Nugteren and Corporaal [2012a]. In this work, we briefly present results for the PolyBench set, demonstrating the practical applicability of algorithmic species in a source-to-source compiler.

⁹ASET can be obtained from: <http://parse.ele.tue.nl/species/>.

¹⁰BONES is available at: <http://parse.ele.tue.nl/bones/>.

Fig. 8. Results for 39 species, showing speed-ups with respect to single-core C code running on the CPU. The number suffixes relate to different species as found within a single algorithm.

The source-to-source compiler BONES provides skeletons for OpenMP, OpenCL, and CUDA, targeting a variety of architectures including GPUs and CPUs. We demonstrate the use of algorithmic species by generating parallel code for the PolyBench suite. As input to BONES we use C program code annotated by `ASET`. We generate OpenMP (CPU), OpenCL (CPU), and CUDA (GPU) code for 39 nonnested species. We use a 4-core Intel Core i7-3770 CPU and a 448 CUDA core NVIDIA GTX470 GPU. We exclude several species from the results, because they show a very limited amount of parallelism and take only a fraction of a millisecond to execute, making start-up and measurement overheads dominant. The results are shown in Figure 8, for which we exclude CPU-GPU data transfer times and normalize against the original C code running on the same CPU.

From the results in Figure 8, we can see the advantages of using an auto-parallelizing compiler: most of the benchmarks show a significant improvement over the original sequential code. Several benchmarks show less or no performance improvement for certain targets. This is mainly caused by factors such as strided memory accesses, bad cache locality, or limited parallelism. Such aspects are either inherent to the application or can be solved by the compiler: the algorithmic species theory provides

all the required details (e.g., locality, reuse, parallelism). For more details on BONES and a comparison with related work, we refer to Nugteren and Corporaal [2012a].

5.4. Using Algorithmic Species for Performance Prediction

We furthermore demonstrate the use of algorithmic species within the *boat hull model*, a performance prediction technique based on the *roofline model* [Williams et al. 2009]. Since programming multi-cores and GPUs can be challenging and time consuming, the boat hull model does not require source code to be available for a given target. To enable performance prediction *prior* to code implementation, the boat hull model generates a specific instance of the *roofline model* for a given algorithmic species. This results in a species-specific roofline model, enabling a first-order performance prediction prior to the development of architecture-specific code. More details on the boat hull model can be found in Nugteren and Corporaal [2012b].

6. EVALUATION AND DISCUSSION

Having introduced and demonstrated the applicability of algorithmic species, we reflect on the goals and requirements as set in Section 2. We first evaluate each of the set requirements individually for algorithmic species.

- (1) Species can be extracted automatically from C program code using ASET, a tool which follows directly from the algorithmic species theory. This removes the need for a time-consuming and error-prone manual classification, making algorithmic species suitable for automatic uses and convenient for manual uses.
- (2) Algorithmic species uses a very limited vocabulary: species are composed of a combination of only five different patterns. The names of these patterns are descriptive, creating an intuitive and easy to use classification. The complete species description forms a high-level abstraction of the mathematical representation used in the polyhedral model. This is not only beneficial for manual uses, but can also benefit automatic uses: for example, this creates opportunities to use species for skeleton-based compilation (e.g., BONES) or for performance prediction purposes (e.g., the boat hull model). For both cases, prior work required the manual identification of some form of algorithm class.
- (3) Notwithstanding intuitiveness, the classification has a formal mathematical basis, derived from the polyhedral model. With this formal basis, every class is formally defined to correspond to its program code and vice versa, which is important for correctness guarantees and to prevent ambiguity or lack of clarity. In contrast, many high abstraction-level classifications and skeleton classifications provide a textual description or merely a few examples to “define” a particular class.
- (4) The classification is not complete in the sense that it will classify any code. Nevertheless, it is complete within certain set limits: any affine loop nest with parallelism can be classified under the algorithmic species theory. This is in contrast to classifications such as pattern languages or algorithmic skeletons, which require new class definitions to be added as different types of algorithms are evaluated.
- (5) Species are fine-grained. They capture information about the structure and amount of parallelism, communication requirements, atomicity, data reuse, and data locality. Algorithmic species are finer-grained than related classifications such as skeletons and idioms. For example, in comparison with Caarls et al. [2006], we add more information, e.g. neighborhood and data sizes and ranges. In contrast, if we compare with alternative code representations such as the polyhedral model, we find that algorithmic species are coarser-grained, capturing only relevant information such as the structure of parallelism.

Fig. 9. Algorithmic species as a common basis for current and future work related to parallel programming.

If we reflect on the examples of listings 3–6, we find that we capture a significant amount of information. For all examples, we capture the amount of data and ranges accessed for all arrays. Additionally, for listing 3, we capture parallelism in both i and j -loops and the fact that there is no data reuse for $M[] []$ and $R[] []$. For listing 4, we capture parallelism in the i -loop, row-wise loads in $M[] []$, and the amount of data reuse in array $v[]$, i.e. it might be beneficial to store or cache $v[]$ locally. For listing 5, we find parallelism in the i -loop and the amount of data reuse in $m[]$. Finally, for listing 6, we find that the i -loop can only be run in parallel in its current form if $r[]$ is written atomically and stored in a shared location.

Since the classification meets our requirements, we strongly believe that it is suitable to meet our goals. In this way, a single platform and programming-language-independent classification can be applied for the various uses of a whole range of existing classifications: algorithmic species can serve as a base for current and future work related to parallel programming (visualized in Figure 9). We evaluate algorithmic species with respect to the two goals as set in Section 2.

- (1) Programmers can use algorithmic species when performing various tasks, e.g. when developing, mapping, porting, optimizing, profiling or debugging code. For example, programmers can use algorithmic species to relate to well-known solutions for certain types of problems or to compare implementations with other programmers. If needed, they can omit details such as access ranges and refer to the combination of access patterns only. In this way, algorithmic species can replace current high-level abstraction classifications such as the Berkeley motifs [Asanovic et al. 2009] and pattern languages [Keutzer and Mattson 2010] by providing the advantages of among others automatic detection of classes and a much richer class information.
- (2) As for automated uses, we have already identified BONES [Nugteren and Corporaal 2012a] and the boat hull model [Nugteren and Corporaal 2012b] as two techniques benefiting from the work on algorithmic species. Still, we envisage more tools and compilers to build upon algorithmic species, creating a common basis to capture essential information of program code such as the structure of parallelism and data locality. For example, algorithmic species can improve work built upon classifications such as skeletons (e.g. Caarls et al. [2006], Campbell [1996], Enmyren and Kessler [2010]), idioms [Carrington et al. 2011], or *Æcute* [Howes et al. 2009], by providing an algorithm classification with advantages over the classifications as outlined in Section 3.

7. FUTURE WORK

We identify opportunities for future work mainly in the direction of extending the classification towards irregular algorithms. Irregular algorithms, i.e. algorithms working on data structures such as sparse matrices, graphs, and trees, are emerging as important computational problems in many domains [Pingali et al. 2011]. If such irregular

algorithms can be classified, insights into the structure of parallelism and data locality of such algorithms could help programmers and compilers produce efficient code.

Apart from extending to irregular algorithms, we believe that algorithmic species can be extended with additional information. This will increase the information embedded in an algorithmic species, but might come at the cost of understandability. Algorithmic species can also be extended with inter-species information, which could for example help compilers and programmers to fuse multiple species.

Finally, we believe it can be beneficial to specify nonparallel species for loop nests that do not meet our dependence check in Equation (19). We can construct different types of nonparallel species, depending on the type of dependences. Such classes can include information regarding potential parallelism in case dependences get resolved (e.g., dynamically or by code rewriting).

8. CONCLUSION

In this work we have introduced algorithmic species, a structured representation of affine loops for both manual and automatic uses. An individual algorithmic species defines a class of algorithms, capturing information such as the structure of parallelism and the amount of data reuse. Species are based on the polyhedral model, a mathematical representation of loop nests, but use vocabulary instead of formalism to create an intuitive classification. In this way, species transform a polyhedral representation into an algorithm classification, offering many advantages over a wide range of existing classifications. In this work, we have set five requirements for an algorithm classification to meet our goals of manual and automated uses. We have evaluated and met these requirements for our classification: (1) species can be extracted automatically from program code, (2) species are intuitive and easy to understand, (3) the individual species are formally defined, (4) all species required to identify parallelism in affine loop nests are pre-defined, and (5) species are fine-grained. We have evaluated existing classifications in detail, but were unable to find one which was able to satisfy these requirements.

We believe that algorithmic species can greatly alleviate many tasks and help overcome parallel computing challenges currently faced by programmers (looking for manual solutions) and compilers (looking for automated solutions). In terms of manual uses, we see algorithmic species as a tool for programmers to ease parallel programming. Programmers can identify types of programming problems, identify parallelism and data reuse, apply known parallel patterns or optimization techniques. Furthermore, programmers can use algorithmic species to communicate, discuss, and describe their computational problems. In terms of automated uses, algorithmic species are a way to facilitate the design and improve the quality of compilers: information embedded in species can help compilers make transformation and parallelization decisions. We have illustrated this by showing the results of a species-based source-to-source compiler, and a species-based performance prediction model. Algorithmic species can thus become a basis for programmers and compilers, creating synergy by bringing the extraction of algorithm details and the definition of classes into a single, common place.

REFERENCES

- ALLEN, R. AND KENNEDY, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. and Syst.* 9, 4, 491–542.
- ASANOVIC K., BODIK, R., DENNEL, J., KEAVENY, T., KEVTZER, K., ET AL. 2009. A view of the parallel computing landscape. *Comm. ACM* 52, 56–67.
- CAARLS, W., JONKER, P., AND CORPORAAL, H. 2006. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*. IEEE.

- CAMPBELL, D. K. G. 1996. Towards the classification of algorithmic skeletons. Tech. rep. YCS 276, University of York.
- CARRINGTON, L., TIKIR, M. M., OLSCHANOWSKY, C., LAURENZANO, M., PERAZA, J., SNAVELY, A., AND POOLE, S. 2011. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the International Conference on Supercomputing*. ACM.
- COLE, M. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press.
- DOLBEAU, R., BIHAN, S., AND BODIN, F. 2007. HMPP: A hybrid multi-core parallel programming environment. In *the 1st Workshop on General Purpose Processing on Graphics Processing Units*.
- ENMYREN, J. AND KESSLER, C. W. 2010. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the 4th International Workshop on High-level Parallel Programming and Applications*. ACM.
- FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Springer Int. J. Parallel Program.* 20, 23–53.
- FULLER, S. H. AND MILLETT, L. I. 2011. Computing performance: Game over or next level? *IEEE Comput.* 44, 31–38.
- HOWES, L., LOKHMOTOV, A., DONALDSON, A., AND KELLY, P. 2009. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Springer.
- KEUTZER, K. AND MATTSO, T. 2010. A design pattern language for engineering (parallel) software. *Intel Technol. J.*
- KONG, X., KLAPPHOLZ, D., AND PSARRIS, K. 1991. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. Parallel Distrib. Syst.* 2, 3, 342–349.
- MASSINGILL, B., MATTSO, T., AND SANDERS, B. 1999. A pattern language for parallel application programming. In *Proceedings of the Sixth Pattern Languages of Programs Workshop*.
- MATTSO, T., SANDERS, B., AND MASSINGILL, B. 2004. *Patterns for Parallel Programming*. Addis.-Wesley.
- MEMBARTH, R., LOKHMOTOV, A., AND TEICH, J. 2011. Generating GPU Code from a high-level representation for image processing kernels. In *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip*.
- NUGTEREN, C. AND CORPORAAL, H. 2011. A modular and parameterisable classification of algorithms. Tech. rep. ESR-2011-02, Eindhoven University of Technology.
- NUGTEREN, C. AND CORPORAAL, H. 2012a. Introducing ‘bones’: a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Workshop on General Purpose Processing on Graphics Processing Units*. ACM.
- NUGTEREN, C. AND CORPORAAL, H. 2012b. The boat hull model: Enabling performance prediction for parallel computing prior to code development. In *Proceedings of the International Conference on Computing Frontiers*. ACM.
- PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M., KALEEM, R., LEE, T., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., ET AL. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd SIGPLAN Conference on Programming Language Design and Implementation*. ACM.
- POUCHET, L.-N., BASTOUL, C., COHEN, A., AND VASILACHE, N. 2007. Iterative optimization in the polyhedral model: part I, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE.
- RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W.-M. W. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*. ACM.
- VERDOOLAE, S. AND GROSSER, T. 2012. Polyhedral extraction tool. In *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques*.
- WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. 2009. Roofline: An insightful visual performance model for multicore architectures. *Comm. ACM* 52, 65–76.
- WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM.
- WOLFE, M. 2010. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*. ACM.

Received June 2012; revised September 2012; accepted November 2012