Type Patterns: Pattern Matching on Shape-Carrying Array Types

Jordy Aaldering
Jordy.Aaldering@ru.nl
Radboud University
Nijmegen, Netherlands

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl
Radboud University
Nijmegen, Netherlands

Bernard van Gastel
Bernard.vanGastel@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

In this paper we present type patterns: a notation for shape-carrying array types that enables the specification of dependent type signatures while maintaining flexibility and a high level of code readability. Similar notations pre-exist, but we extend them to support rank-polymorphism and specifications of arbitrarily complex constraints between values and types. Furthermore, we enable type patterns to double as a pattern matching mechanism against shapes and shape-components of array arguments, making those values directly available in the corresponding function bodies.

While this notation could be used as a basis for a dependently typed language, in our prototypical implementation in the context of SaC we do not require all dependencies to be resolved statically. Instead, we follow a hybrid approach: we map the proposed type patterns into the pre-existing type system of SaC, and we generate additional constraints which we try to statically resolve as far as possible by means of partial evaluation. Any remaining constraints are checked at run-time. We outline our implementation in the context of the SaC ecosystem, and present several examples demonstrating the effectiveness of this hybrid approach based on partial evaluation.

KEYWORDS

Array Programming, Single assignment C, Rank-Polymorphism, Hybrid Types, Dependent Types, Type Constraints, Partial Evaluation, Pattern Matching, Shape Pattern, Type Pattern

ACM Reference Format:
Jordy Aaldering, Sven-Bodo Scholz, and Bernard van Gastel. 2023. Type Patterns: Pattern Matching on Shape-Carrying Array Types. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652561.3652572

1 INTRODUCTION

Array programming languages play a crucial role in a wide range of data-centric applications, where the manipulation of multi-dimensional arrays is at the core of many computational tasks. In this domain, understanding and controlling the rank and shape of arrays is crucial, as they directly influence the correctness and efficiency of algorithms.

While array languages traditionally strive for universal applicability of operators to arrays of arbitrary rank and shape, some minimal restrictions on argument domains are usually inevitable, be it to avoid out-of-bound accesses, or to ensure some other structural consistency. Unfortunately, the nature of these constraints is typically not only related to ranks and shapes of arguments alone, but to the values of arguments and return types as well. The most prominent example for such a situation is element selection, which requires the index argument to have a value that is within the bounds of the shape of the array argument.

Using advanced type systems to provide static guarantees for such domain restrictions is highly desirable. It makes these constraints explicit, provides the means to mechanically identify violations of them, and it opens the door for better code optimisation. The value-dependent nature of these constraints, in the context of array programming languages, has led to several different type systems [21, 31, 35, 39, 40] that try to balance the trade-offs between readability of code, decidability of the type system, and expressiveness of the language. Although most of these approaches prioritise decidability, in this paper we take a different approach. Rather than starting out from a type system, we start out from a notation for domain constraints that aims at programming productivity. Readability of domain constraints without any restrictions in the expressiveness of the language is the primary goal of this work.

We introduce variables into a notation for shape-carrying type signatures that can be seen as pattern matching constructs for array shapes. This enables programmers to conveniently refer to argument shapes and shape-components in function bodies through these variables. Predicates on these variables can specify arbitrary relationships between argument domains and result co-domains. Similar notations exist in prior type-based work, such as [21, 39], but our approach takes the idea further by introducing support for rank-polymorphism and by enabling specifications of arbitrarily complex relations between domains and co-domains.

Given the expressiveness of these type patterns, a full implementation within the context of a type system would for many practical examples raise decidability issues. Several programs would require explicit assertions on program inputs, other programs would require additional proofs to aid the type system in proving static correctness. Practical experience in the context of fully dependently typed languages such as Agda [6] demonstrates that such proofs typically require non-trivial modelling which would be incompatible with our quest for readability and programming productivity.

Therefore, we suggest to map our type patterns into a less powerful type system where we generate explicit constraints, that either can be resolved using partial evaluation techniques such as [5, 23], or that can produce run-time checks and errors, which aims to assist programmers in identifying incompatibilities at the earliest possible stage. We chose the type system of Single assignment C

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 bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IFL 2023, August 29–31, 2023, Braga, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1631-7/23/08.
https://doi.org/10.1145/3652561.3652572

(SaC) [16, 17] as our target here, since it already builds on the ideas of partial evaluation. We expand on previous work [18, 33] which investigates how constraints on array shapes can be introduced.

Whilst type patterns share similarities with dependent types, the choice of static or dynamic checking makes the two clearly distinct. Whereas dependent types consider statically provable programs as their goal, our goal is to maximise programmer productivity. One might then see type patterns as a form of hybrid type checking, but that would still require a type formalism that captures type patterns in their entirety which we intentionally avoid.

Type patterns also share similarities with pattern matching. Whilst pattern matching can be applied generally, we focus on array properties, specifically at rank and shape-components. In contrast to the pattern matching found in most functional languages, we allow the same variable to be defined multiple times in order to denote constraints between those ranks and shapes. Another aspect that sets type patterns apart from conventional pattern matching, is the fact that type patterns occur within the type signatures of functions rather than within function bodies.

Since type patterns do not fit into any of these categories, one might then believe that they are simply syntactic sugar. But again, this is not the case. Whilst type patterns are indeed rewritten to pre-existing code during compilation, they are non-trivially woven into the program in order to provide as much optimisation and static analysis as possible, and enabling the automatic generation of descriptive error messages explicitly relating to the type patterns defined in the source code.

Our contributions are:

- Type Patterns as an amalgamation of type specification and pattern matching, in the context of functions on arrays.
- Several examples demonstrating the readability, expressiveness, and effectiveness of the proposed approach.
- A formal mapping from type patterns into the pre-existing type system of SaC.
- A formal mapping from type patterns into pre- and post-conditions, as well as into code that actually performs pattern matching on argument and return value shapes.
- A sketch of an implementation in the context of the SaC compiler ecosystem, providing details on how the constraints are woven into the data-flow, enabling static feedback through partial evaluation.
- An implementation of type patterns in the SaC compiler[1] and its standard library[2].

2 SINGLE ASSIGNMENT C

We introduce type patterns in the context of the SaC programming language. SaC is a functional array language that, as the name suggests, resembles the syntax of imperative languages such as C, whilst remaining side-effect-free [16, 17]. Although SaC programs might look imperative, assignments are considered cascading let expressions, whereas loops are implemented as tail-recursive functions. One of the distinguishing features of SaC is its support for rank-polymorphism, i.e. the ability to define functions that take arrays of arbitrary unknown rank as arguments. In SaC all data structures are considered arrays, making it a fitting target for our implementation of type patterns.

[1] https://sac-home.org
[2] https://github.com/SacBase/Stdlib

2.1 Array types

Arrays in SaC are internally represented by three values: an integer describing the rank of the array, a vector of integers containing the length along each axis, i.e. the shape, and a vector containing all data values in a flattened notation. Table 1 shows some examples of arrays, along with their internal representations.

Array		
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	rank: shape: data:	2 [3,3] [1,2,3,4,5,6,7,8,9]
[1, 2, 3, -4, -5]	rank: shape: data:	1 [5] [1,2,3,-4,-5]
[[1, 2, 3]]	rank: shape: data:	2 [1,3] [1,2,3]
0.5	rank: shape: data:	0 [] [0.5]

Table 1: Array representation in SaC

Note that the rank is always equal to the length of the shape vector, and that the product of the shape vector is equal to the total number of elements in the array.

This array information will always become fully available during run-time, but might already be partially known during compile-time. Namely, we might be able to infer statically the rank, shape, or even the values of an array. We distinguish between four cases of statically known information: either we have an Array of Known Values (AKV), an Array with a statically Known Shape (AKS), an Array where we only Know the Dimensionality (AKD), or we have an Array where even the Dimensionality is Unknown and we have no static knowledge at all (AUD). Figure 1 shows the hierarchy of these types, using integer arrays as an example. See [29] for further reading.

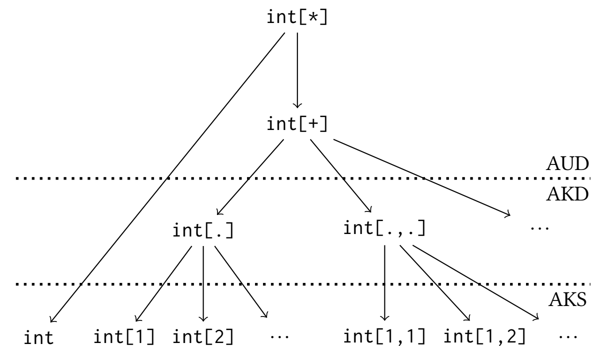

Figure 1: Hierarchy of integer array types in SaC

2.2 Domain constraints

In order to more efficiently check domain constraints imposed by primitive operations on arrays, such as array selection, SaC offers a hybrid approach [23]. During compilation it inserts checks around primitive functions, such as around the built-in element selection primitive `_sel_VxA_`. These inserted checks might be statically resolved based on the values, shapes, or ranks of the input. Depending on whether the arguments are AKV, AKS, or AKD, certain checks can be statically resolved.

Consider this element selection example. This built-in function expects an array and an index vector with as many elements as the rank of that array. This index vector must be non-negative and each element must be less than the corresponding shape-element of the array. It always returns the scalar value at that exact index, and does not allow for the selection of a larger sub-array. We require two constraints: one to ensure that the length of the selection vector is equal to the rank of the array, and one to check that each value in the selection vector is in bounds of the shape of the array. Here we can statically resolve the first constraint if the index vector is AKS and the array is only AKD. Whereas to statically resolve the second constraint we require that the index vector is AKV and the array is AKS.

2.3 Tensor comprehension

SaC allows for rank- and shape-invariant programming by means of 'tensor comprehensions' [30]. Essentially, tensor comprehensions are a mapping from index vectors to values. The result of a tensor comprehension is an array, whose shape depends on the range of these index vectors. For example, we can use tensor comprehensions to increment all values in the variable-rank array `arr`.

```
inc = { iv -> arr[iv] + 1 };
```

This index vector `iv` is a variable-length vector, where the length is based on the usages of `iv`. If `arr` is for example an `N×M` matrix, `iv` is a two-element index vector that is repeatedly applied to the increment expression, with values ranging from `[0,0]` to `[N-1,M-1]`.

The range of the index vector can usually be inferred automatically from the uses of `iv`, but may also be defined manually. For example, we might only want to apply the tensor comprehension to a sub-array of `arr`.

```
shp = shape(arr) / 2;
inc = { iv -> arr[iv] + 1 | iv < shp };
```

Here we denote that `iv` ranges up until and excluding the vector `shp`. Because no lower bound is given, it is set to a zero-element vector of the same rank as `shp`. Note that the shape of the result `inc` depends on the range of the index vector `iv` and not on the array `arr`, the shape of `inc` will thus be `shp` and not `shape(a)`.

3 INTRODUCING TYPE PATTERNS

We introduce type patterns from the perspective of introducing a pattern matching mechanism, as this matches more directly with the way we map them into the pre-existing hybrid type system of SaC. An alternative way to think about type patterns is to see them as ways to introduce types, through the introduction of variables within type signatures of function definitions.

3.1 Syntax of type patterns

We start out with the syntax of type patterns, shown in Figure 2.

```
<pattern>  ::= <type> '[' <feature> (',' <feature>)* ']'

<feature>  ::= <single>
             |  <multiple>

<single>   ::= '.'
             |  <num>
             |  <id>

<multiple> ::= '*'
             |  '+'
             |  <num> ':' <id>
             |  <id> ':' <id>
```

Figure 2: EBNF for type patterns

This syntax extends of the notion of array types from SaC, providing more syntactical flexibility and introducing variables within these types. Similar to the types in SaC, type patterns consist of an element type followed by a shape specification in square brackets. The shape specification is a pattern, which consists of a list of 'features'. These features either describe a single rank of the argument, or a larger slice (a sub-vector) of its shape vector.

Single dimensions are captured by the `<single>` rule in Figure 2 and can be denoted by the '`.`' symbol, a constant number, or a variable name. A fixed number requires the length of the corresponding rank component to match that very number, whereas the '`.`' symbol allows for an arbitrary extent. If instead a variable is being used, the extent found needs to match all other occurrences of that very variable within a given function definition. For example, consider the type pattern `int[n,n]`. It allows for square matrices of arbitrary size $n \times n$ where `n` is a non-negative integer value.

Rank-polymorphism requires the ability to denote a statically unknown number of ranks. To cater for this, the proposed type patterns support features that capture slices of a given shape. These are captured by the `<multiple>` rule in Figure 2. We distinguish four cases, depending on the rank of the slice we want to match, and whether we are interested in the slice itself or not. If we are not interested in the slice itself, we can use one of the symbols '`+`' and '`*`'. They match any slice of at least rank 1 or 0, respectively.

If we are interested in the slice itself, the feature consists of two components separated by a '`:`' symbol. The first component relates to the rank of that slice, whereas the second component relates to the slice itself, i.e. the lengths of the corresponding axes. For the rank component, similar to the single-rank features, we can either use a fixed number or an identifier. For example, `int[3:shp]` matches rank 3 arrays only, and matches the entire shape vector against the variable `shp` that will thus be a three-element vector. Note here that variables after the '`:`' symbol, such as `shp`, denote slices and therefore always represent vectors, even if they match no ranks (empty vector) or individual ranks (singleton vector). For example, a type pattern `int[1:n,1:n]`, similar to the `int[n,n]` example above, matches arbitrary sized square matrices, but `n` now is a singleton vector rather than a scalar value.

Finally we have the case where the rank of the slice is also a variable, enabling a match of a variable-rank slice of an array shape. Here, the first component matches against the rank of the slice, and the second component again matches against the slice itself. For example, the pattern `int[d:shp]` matches arrays of arbitrary shape, and binds the variable `d` to the rank of the array, and the variable `shp` to the shape of the array.

These features can be combined to create complex type patterns. Consider for example a type pattern `int[5,n,d:shp] v`. It matches any shape of at least rank 2, whose extent in the first axis is exactly five, and whose extent in the second axis is `n`. Any potentially remaining axes are bound to the vector `shp`. These variables (`n`, `d`, and `shp`) can also be used in the function body. Following are some examples for arguments `v` and the resulting values of `n`, `d`, and `shp`.

shape(v)	5	n	d	shp
[5,1]	5	1	0	[]
[5,1,7]	5	1	1	[7]
[5,0,1,2]	5	0	2	[1,2]
[5,2,1,2,3,4]	5	2	4	[1,2,3,4]

Variable names may be used more than once. For example, the variable `n` can be reused in a variable-rank feature `[5,n,n:shp] v`. This pattern matches arrays of at least rank 2 whose extent in the second axis determines the number of ranks that follow. Some examples of matching shapes are:

shape(v)	5	n	shp
[5,0]	5	0	[]
[5,1,0]	5	1	[0]
[5,1,42]	5	1	[42]
[5,4,1,2,3,4]	5	4	[1,2,3,4]

Note that `d` in the first example, and `n` in the second example, are equal to the length of `shp`, which is always the case for such variable-rank patterns.

3.2 Using type patterns

Making use of these type patterns, we can extract the shape and the rank of an argument without requiring the use of built-in functions, allowing for simple re-definitions of the functions `dim` and `shape`:

```
int
dim(int[d:shp] arr)
{
    return d;
}
```

```
int[d]
shape(int[d:shp] arr)
{
    return shp;
}
```

In the example of `shape` we can see how the type pattern notation allows for an intuitive way to express the relation between the rank of the argument and the shape of the result.

Type patterns also allow us to impose constraints between multiple arguments and return values of a function. For example, the function signature of shape-polymorphic addition of two floating point arrays can be specified by:

```
float[d:shp]
add(float[d:shp] a, float[d:shp] b)
{
    return { iv -> a[iv] + b[iv] };
}
```

Here the type patterns in the signature demand both arguments to have the exact same shape. Additionally it specifies that the shape of the result will also be equal to the shapes of those two inputs.

An example of a more intricate shape relation between argument and result shapes can be seen in a simplified definition of the `take` function:

```
float[n:shp,m:inner]
take(int[n] shp, float[n:outer,m:inner] arr)
{
    return { iv -> arr[iv] | iv < shp };
}
```

We see here that the result has the same rank as the second argument, where the first `n` elements of the result shape are defined through the values of the first argument. These type patterns demonstrate an interesting case: when looking at the type pattern of argument `arr` in isolation, we have two features that match against a variable number of ranks captured by the variables `n` and `m`. This is potentially ambiguous. However, in the given context, we have a uniquely determined constraint for `n` through the shape of the first argument.

This potential ambiguity is inevitable as soon as we have more than one variable-rank feature within a type pattern. Fortunately, these cases can be identified and rejected statically, during the translation of type patterns into explicit constraint checking code.

Finally we consider the shape-polymorphic selection function, which uses the `_sel_VxA_` primitive (applied using `arr[...]`) to select a single scalar, or a larger sub-array, from a given array. This function will serve as our running example throughout the remainder of this paper.

```
int[d:shp]
sel(int[n] idx, int[n:outer,d:shp] arr)
{
    return { iv -> arr[idx ++ iv] | iv < shp };
}
```

Again, we see here a potential ambiguity between the two variable-rank features of argument `arr`, captured by the variables `n` and `d`. This ambiguity is resolved through the occurrence of `n` within the type pattern of the first argument.

4 TRANSFORMING TYPE PATTERNS

Rather than mapping type patterns and their constraints into some form of dependent types, we instead map them into the existing type system of SaC along with a set of constraints that are inserted into the program. Revisiting the signature of the selection function from the previous section, we can see that it combines multiple variable-rank features, across multiple argument and return types.

```
int[d:shp]
sel(int[n] idx, int[n:outer,d:shp] arr)
```

The rank and shape of the result of this function must match `d` and `shp` respectively, which are defined by the type pattern of argument `arr`. However, that argument has two variable-rank features, captured by the variables `n` and `d`. In this case the value of `n` is required before the value of `d` can be computed. We compute `n` from the length of the index vector `idx`, after which we are able to compute the values of `d` and `shp`, which finally provides us with the rank and shape of the result.

From this example it becomes clear that the order in which type patterns are resolved matters. We might need to switch between multiple arguments and features in order to be able to resolve complicated and intertwined dependencies. To cater for this, we consider type patterns as a constraint resolution problem. We do so by not only generating the relevant constraints for each feature, but by also including the dependencies that need to be resolved, i.e., the values we need before that constraint can be resolved.

4.1 Type pattern analysis

As a first step, we convert type patterns to SaC types. Looking at the selection example, the index vector is a rank-one array of length n, but because we do not statically know n we have no choice but to convert it to the SaC type `int[.]`. The array argument consists of two variable-rank features, both of which could be empty, thus now the only fitting SaC type is `int[*]`. Similarly for the return value, which has a single variable-rank feature of which we have no static knowledge. Consequently, we need to map the type patterns of the selection function to:

```
int[*]
sel(int[.] idx, int[*] arr)
```

We formalise this process by means of a recursive transformation function: Analyse Type Patterns (ATP).

$$\mathsf{ATP}(\mathit{type_pattern},\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim}) = (\mathit{sac_type},\ \mathsf{kshp}',\ \mathsf{kdim}',\ \mathsf{vdim}')$$

It traverses through the given type pattern, feature by feature, collects information about the features seen so far, until it returns a SaC type along with the information accumulated in three additional parameters:

(1) kshp: a list of known shape components. We denote these by a list of integers enclosed in square brackets, and we start ATP with `[]`.
(2) kdim: an integer for the minimal number of ranks required, starting with 0.
(3) vdim: a list containing the rank identifiers of variable-rank features (such as `*` and `d:shp`). We denote these by a list of identifiers enclosed in square brackets, starting with `[]`.

The different cases of ATP directly follow the different options for pattern features. When the feature is either an individual dot symbol or a variable, we increment kdim to reflect the need for exactly one dimension.

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[\texttt{.}, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim}+1,\ \mathsf{vdim})\end{aligned}$$

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[\mathit{id}, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim}+1,\ \mathsf{vdim})\end{aligned}$$

In our rule, we use tl to denote the (possible empty) remaining features of the type pattern which are needed for the recursive step. Similarly for integers n, but since that value describes the extent of a rank, we also add that integer to the list of known shapes. This is the only case where the number of known shapes is changed.

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[\mathsf{n}, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp} \mathbin{++} [\mathsf{n}],\ \mathsf{kdim}+1,\ \mathsf{vdim})\end{aligned}$$

If instead we match on a shape-component with some constant non-negative integer rank n, that feature describes the shape-component of the following n ranks, so we increase the number of known ranks by that amount. We ignore the shp identifier because it has no impact on the resulting type.

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[\mathsf{n}\!:\!\mathit{shp}, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim}+\mathsf{n},\ \mathsf{vdim})\end{aligned}$$

When this rank is instead an identifier d, we append that identifier to the list of variable ranks. We do not increase the number of known ranks, because d is potentially zero.

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[d\!:\!\mathit{shp}, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim} \mathbin{++} [d])\end{aligned}$$

Finally, when we encounter a `*` we simply add it to the list of variable ranks as well. We have a similar case if we encounter a `+`, but since it describes a shape with a non-zero rank, we also increment the number of known ranks.

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[*, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim} \mathbin{++} [*])\end{aligned}$$

$$\begin{aligned}&\mathsf{ATP}(\mathit{type}[+, \mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})\\ &= \mathsf{ATP}(\mathit{type}[\mathit{tl}],\ \mathsf{kshp},\ \mathsf{kdim}+1,\ \mathsf{vdim} \mathbin{++} [*])\end{aligned}$$

Once the function is applied to all features, the resulting SaC type can be determined:

$$\mathsf{ATP}(\mathit{type}[\,],\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim}) = (T,\ \mathsf{kshp},\ \mathsf{kdim},\ \mathsf{vdim})$$

where

$$T = \begin{cases} \mathit{type}[*] & \text{if } |\mathsf{vdim}| > 0 \text{ and } \mathsf{kdim} = 0 \\ \mathit{type}[+] & \text{if } |\mathsf{vdim}| > 0 \text{ and } \mathsf{kdim} > 0 \\ \mathit{type}[\texttt{.}_1, \ldots, \texttt{.}_n] & \text{if } |\mathsf{vdim}| = 0 \text{ and } \mathsf{kdim} = n > |\mathsf{kshp}| \\ \mathit{type}[\mathsf{kshp}] & \text{if } |\mathsf{vdim}| = 0 \text{ and } \mathsf{kdim} = |\mathsf{kshp}| \end{cases}$$

If vdim is not empty, the type has an unknown rank that is at least of length kdim. Otherwise, we statically know that the rank of the array is equal to kdim. Furthermore, if the number of known shapes and ranks are equal, the type pattern only contains constant rank sizes, such as `int[5,3,7]`, and thus the shape is statically known.

As an example we consider a type pattern `int[5,n,d:shp]`. Applying ATP to it gives:

$$\begin{aligned}&\mathsf{ATP}(\texttt{int[}\mathbf{5}\texttt{,n,d:shp]},\ 0,\ 0,\ [\,])\\ &= \mathsf{ATP}(\texttt{int[}\mathbf{n}\texttt{,d:shp]},\ 1,\ 1,\ [\,])\\ &= \mathsf{ATP}(\texttt{int[}\mathbf{d{:}shp}\texttt{]},\ 1,\ 2,\ [\,])\\ &= \mathsf{ATP}(\texttt{int[]},\ 1,\ 2,\ [d])\\ &= (\texttt{int[+]},\ 1,\ 2,\ [d])\end{aligned}$$

This results in a known rank of at least two, but because there are also variable ranks the resulting SaC type is `int[+]`.

4.2 Constraint generation

The next step is to aggregate all possible constraints that result from type patterns. We use a similar approach as in the previous section, recursing through the features of a type pattern and collecting information of the resulting constraints. The basic idea is to accumulate an environment of constraints for all variables.

This constraint aggregation poses two particular challenges. Firstly, we may find constraints for a single pattern variable within more than one type pattern of a function signature. This implies that we cannot look at single type patterns in isolation. We resolve this issue by collecting an environment of constraints and by passing this environment from one type pattern within a function signature to the next. Secondly, we can have dependencies between constraints: As soon as there is a variable-rank feature combined with any other feature, any resolution process can no longer be done in arbitrary order. We record such dependencies as we generate the environment of constraints.

We denote our constraint environment by entries of the form:

$$var \mapsto [\langle expr_1,\ deps_1\rangle,\ \ldots,\ \langle expr_n,\ deps_n\rangle]$$

where var refers to any variable name occurring in a given function signature, be it in a type pattern or an argument name itself. Each tuple $\langle expr_i,\ deps_i\rangle$ represents a SaC expression $expr_i$ that constitutes the constraint for var. I.e. at runtime, the value of var needs to be identical to the value of that expression, and $deps_i$ contains the set of variables that stem from variable-rank features and are referred to within $expr_i$.

For example, for a function argument `int[d,d:shp] v`, we want to infer the following two environment entries:

$$\{\ d \mapsto [\langle \mathtt{shape}(v)[0],\ \emptyset\rangle,\ \langle \mathtt{dim}(v) - 1,\ \emptyset\rangle]$$
$$shp \mapsto [\langle \mathtt{take}(d,\ \mathtt{drop}(1,\ \mathtt{shape}(v))),\ \{d\}\rangle]\ \}$$

For inferring such constraint environments, we define a function: Generate Type Constraints (GTC).

$$\mathtt{GTC}(type_pattern,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \sigma'$$

It is being successively applied to all type pattern of any given type signature and accumulates a constraint environment σ for the entire signature while doing so. In addition to the current type pattern and the constraint environment to be generated, it uses two further parameters when being applied to each individual type pattern within a function signature. These parameters are:

(1) `cdim`: the current position in the shape vector of the argument, starting with 0.
(2) `deps`: a set of variable-rank identifiers encountered so far.

Following are the recursive cases of the GTC function. In the case that we encounter a dot, we only need to increment the current position in the shape. Because in this case we do not care about the result, no constraints are added to the environment.

$$\mathtt{GTC}(type[.,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+1,\ \mathtt{deps},\ \sigma)$$

For constant integers n we similarly increment the current shape position. However now we also add a constraint to the environment of the argument with the current dependencies. This constraint gets the extent of the current rank from the current shape position, and checks that it is equal to n. We write $\sigma[var \mapsto \langle expr,\ deps\rangle]$ to denote that the environment σ is extended with a new entry $\langle expr,\ deps\rangle$ for the constraints list of var.

$$\mathtt{GTC}(type[\mathtt{n},tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+1,\ \mathtt{deps},\ \sigma[v \mapsto \langle \mathtt{n} == \mathtt{shape}(v)[\mathtt{cdim}],\ \mathtt{deps}\rangle])$$

We have a similar case if we encounter an identifier id. However we now add a constraint to the environment for id instead.

$$\mathtt{GTC}(type[id,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+1,\ \mathtt{deps},\ \sigma[id \mapsto \langle \mathtt{shape}(v)[\mathtt{cdim}],\ \mathtt{deps}\rangle])$$

In the case of a shape of fixed length n, we instead increase the current shape position by this fixed amount. We also add a constraint for the shape identifier shp with the current dependencies. This constraint drops the first `cdim` dimensions from the shape, and then takes the following n elements to get the shp vector.

$$\mathtt{GTC}(type[\mathtt{n}{:}shp,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+\mathtt{n},\ \mathtt{deps},\ \sigma[shp \mapsto \langle \mathtt{take}(\mathtt{n},\ \mathtt{drop}(\mathtt{cdim},\ \mathtt{shape}(v))),\ \mathtt{deps}\rangle])$$

For variable-rank features we have a somewhat similar case. Instead of increasing the current position by a fixed value, we increase it by d. Since d is a variable-rank identifier we add it to the accumulated dependencies. We also have a similar constraint for shp, where instead we take d elements. Here again because d is a variable-rank identifier, we add it to the dependencies of this constraint.

Additionally a constraint for d itself is required. We get this constraint from the dimensionality of v, by subtracting from it the minimal rank of v found by ATP (v.kdim) as well as the variable-rank identifiers found by ATP (v.vdim), excluding d. We compute this as $\mathtt{dim}(v) - v.\mathtt{kdim} - \mathtt{sum}(v.\mathtt{vdim}) + d$, which we hereafter shorten to $\mathtt{dimcalc}(v,\ d)$.

$$\mathtt{GTC}(type[d{:}shp,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+d,\ \mathtt{deps}\cap\{d\},\ \sigma[d \mapsto \langle \mathtt{dimcalc}(v,\ d),\ v.\mathtt{vdim}\setminus\{d\}\rangle][shp \mapsto \langle \mathtt{take}(d,\ \mathtt{drop}(\mathtt{cdim},\ \mathtt{shape}(v))),\ \mathtt{deps}\cap\{d\}\rangle])$$

Finally we have the cases for * and +. In the case of * there is nothing more to do, and we continue with the next feature. If instead we encounter a +, we increment the current shape position before continuing.

$$\mathtt{GTC}(type[*,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma)$$

$$\mathtt{GTC}(type[+,tl]\ v,\ \mathtt{cdim},\ \mathtt{deps},\ \sigma) = \mathtt{GTC}(type[tl]\ v,\ \mathtt{cdim}+1,\ \mathtt{deps},\ \sigma)$$

In certain scenarios it is possible that these generated constraints cause an out-of-bounds selection. Consider a variable-rank feature where the input is expected to be of at least rank three. Since ATP will convert this type pattern to the type `int[+]`, a rank one or two input will not be seen as a type error by the type-checker. As a result, such a lower rank input will instead cause an out-of-bounds exception in the generated code when trying to construct features that expect an input of at least rank three. To avoid generating erroneous code, we additionally add a constraint to ensure that the rank of the argument is large enough to fit the given type pattern.

This is done by comparing the rank of the argument v to the minimal rank that was found by ATP (v.kdim). If there are no variable ranks we check whether the rank is exactly equal to the number of known ranks (1), otherwise we check whether the argument has at least that many ranks (2).

This results in the following base case:

$$\texttt{GTC}(type[\,]\ v,\ \texttt{cdim},\ \texttt{deps},\ \sigma) = \begin{cases} \sigma[v \mapsto \langle \texttt{dim}(v) == v.\texttt{kdim},\ \emptyset\rangle] & \text{if vdim} = \emptyset \quad (1) \\ \sigma[v \mapsto \langle \texttt{dim}(v) >= v.\texttt{kdim},\ \emptyset\rangle] & \text{otherwise} \quad (2) \end{cases}$$

Consider again the example `int[5,n,d:shp]` with a mapping σ that might have been populated by type patterns of previous arguments. In each step GTC resolves a single feature from the type pattern. This results in an updated environment with a constraint for each identifier, along with a constraint that checks the rank of the argument.

$$\begin{aligned}
&\texttt{GTC}(\texttt{int[}\mathbf{5}\texttt{,n,d:shp]}\ v,\ 0,\ \emptyset,\ \sigma) \\
&= \texttt{GTC}(\texttt{int[}\mathbf{n}\texttt{,d:shp]}\ v,\ 1,\ \emptyset, \\
&\qquad \sigma[v \mapsto \langle 5 == \texttt{shape}(v)[0],\ \emptyset\rangle]) \\
&= \texttt{GTC}(\texttt{int[}\mathbf{d:shp}\texttt{]}\ v,\ 2,\ \emptyset, \\
&\qquad \sigma[v \mapsto \langle 5 == \texttt{shape}(v)[0],\ \emptyset\rangle] \\
&\qquad\ [n \mapsto \langle \texttt{shape}(v)[1],\ \emptyset\rangle]) \\
&= \texttt{GTC}(\texttt{int[]}\ v,\ 2 + d,\ \{d\}, \\
&\qquad \sigma[v \mapsto \langle 5 == \texttt{shape}(v)[0],\ \emptyset\rangle] \\
&\qquad\ [n \mapsto \langle \texttt{shape}(v)[1],\ \emptyset\rangle] \\
&\qquad\ [d \mapsto \langle \texttt{dimcalc}(v,\ d),\ \emptyset\rangle] \\
&\qquad\ [shp \mapsto \langle \texttt{take}(d,\ \texttt{drop}(2,\ \texttt{shape}(v))),\ \{d\}\rangle]) \\
&= \sigma[v \mapsto \langle 5 == \texttt{shape}(v)[0],\ \emptyset\rangle] \\
&\quad [n \mapsto \langle \texttt{shape}(v)[1],\ \emptyset\rangle] \\
&\quad [d \mapsto \langle \texttt{dimcalc}(v,\ d),\ \emptyset\rangle] \\
&\quad [shp \mapsto \langle \texttt{take}(d,\ \texttt{drop}(2,\ \texttt{shape}(v))),\ \{d\}\rangle] \\
&\quad [v \mapsto \langle \texttt{dim}(v) >= 2,\ \emptyset\rangle]
\end{aligned}$$

5 CONSTRAINT RESOLUTION

We take a partial evaluation approach based on the idea of symbiotic expressions [5]. Symbiotic expressions provide a method for algebraic simplification, based on expressions inserted into the code. We take a similar approach, by inserting the generated constraints into the program. Similarly to symbiotic expressions, the insertion of this code can potentially lead to further optimisation. Unlike symbiotic expressions, we propose that the inserted code is not necessarily removed after optimisation, in order to allow for run-time errors for constraints that could not statically be resolved.

5.1 Resolution algorithm

Now that we have the necessary constraints for all arguments and features of a function signature, the next step is to transform each of those constraints to either a variable assignment, or to a boolean check expression. As discussed previously, the order in which the code for these constraints is inserted into the program is important. We use the dependencies found by GTC to derive the correct ordering. Following is the description of an algorithm that uses the dependencies to generate an ordered list of assign statements and check expressions, ensuring that the dependencies of those constraints are resolved in the correct order.

We require two additional sets for this algorithm:

(1) `defined`: a set which contains all identifiers that have thus far been defined. Initially, these are only the arguments themselves.
(2) `exists`: a set which contains all identifiers that have remaining constraints. These are initially all identifiers that occur in the type patterns of the function signature.

This results in two ordered lists for the generated assignments and checks. These two lists will be inserted into the code as described in section 5.2. The algorithm is repeated until no more code can be generated. At that point either every constraint has been resolved, or there are multiple constraints that cannot be resolved because they depend on each other in some way, in which case we raise an error and print the constraints that could not be resolved.

```
changed = true;

while changed:
  changed = false

  for id in exists:
    all_dependencies_resolved = true

    constraints = env[id]
    for entry in constraints:
      deps, expr = entry
      if union(deps, defined) is not empty:
        all_dependencies_resolved = false
        continue

      if id is not in defined:
        assignments += create_assign(id, expr)
        defined += {id}
      else:
        checks += create_check(id, expr)

      env[id] -= {entry}
      changed = true

    if all_dependencies_resolved:
      exists -= {id}
```

Figure 3: Pseudo-code for the resolution algorithm

The algorithm described in Fig. 3 iterates over all identifiers in the `exists` set, which are the identifiers that have remaining constraints. It then iterates over the remaining constraints of those identifiers. If a constraint has dependencies that are not yet defined, then this constraint still has unresolved dependencies and is skipped. Otherwise there is a case distinction depending on whether the identifier itself has a definition yet. If not, an assignment to this identifier is generated from the value of the constraint and the identifier is added to the `defined` set. Otherwise there already exists such a definition, and instead a check is generated to ensure that this definition is equal to the value of the constraint.

Finally the entry is removed from the environment and we continue with the next constraint. After no more changes occur, if `exists` is empty all constraints were able to be resolved, otherwise we have an ambiguous case and `exists` contains the remaining constraints that could not be resolved due to circular dependencies.

To illustrate such an ambiguous case we take another look at the selection function example. If we consider the array argument `int[n:outer,d:shp]` in isolation, we would receive the following environment σ from GTC:

$$\begin{aligned} \{\ \underline{n} &\mapsto [\langle \mathsf{dimcalc}(arr,\ n),\ \{\underline{d}\}\rangle] \\ outer &\mapsto [\langle \mathsf{take}(n,\ \mathsf{drop}(0,\ \mathsf{shape}(arr))),\ \{n\}\rangle] \\ \underline{d} &\mapsto [\langle \mathsf{dimcalc}(arr,\ d),\ \{\underline{n}\}\rangle] \\ shp &\mapsto [\langle \mathsf{take}(d,\ \mathsf{drop}(n,\ \mathsf{shape}(arr))),\ \{n, d\}\rangle]\ \} \end{aligned}$$

Here we immediately see an example of an ambiguous case that cannot be resolved by our algorithm, as `n` depends on `d` and vice versa. Already in the first iteration of the algorithm there is nothing that can be done, since each constraint has remaining dependencies. However if we include the index vector argument `int[n]`, a new entry is added in the environment for identifier `n`:

$$\begin{aligned} \{\ n &\mapsto [\langle \mathsf{dimcalc}(arr,\ n),\ \{d\}\rangle, \\ &\qquad \underline{\langle \mathsf{shape}(idx)[0],\ \emptyset\rangle}] \\ outer &\mapsto [\langle \mathsf{take}(n,\ \mathsf{drop}(0,\ \mathsf{shape}(arr))),\ \{n\}\rangle] \\ d &\mapsto [\langle \mathsf{dimcalc}(arr,\ d),\ \{n\}\rangle] \\ shp &\mapsto [\langle \mathsf{take}(d,\ \mathsf{drop}(n,\ \mathsf{shape}(arr))),\ \{n, d\}\rangle]\ \} \end{aligned}$$

In this case the algorithm is able to generate an assignment for this new constraint of `n`, because it has no dependencies. This results in `n` being added to the `defined` set. Now that `n` is defined, the constraint for `d` can be resolved and an assignment to `d` is generated. Afterwards, the identifiers `n` and `d` are both defined and all remaining constraints can be resolved: an assignment is generated for `outer` and `shp`, and because `n` already exists a check is generated for the remaining constraint of `n`.

5.2 Code generation

At this point these ordered lists of generated assignments and checks can be inserted into the program. Special care needs to be taken to ensure that the generated code is inserted at the most optimal location in the program.

To provide the compiler with as much static information as possible, the generated assignments and checks should preferably be inserted around the call site of the corresponding function, where typically more information about the rank, shape, or values of the arguments is statically available. This makes it more likely that the checks can statically be resolved, allowing for more type errors at compile time and reducing overhead at run-time. However a static dispatch is not always possible, such as for overloaded functions. In this case we have no choice but to insert the generated code into the function body instead.

To cater for both cases with a single implementation, we rely on the inlining capabilities of the compiler. Inlining ensures that if a call site of a function has been statically dispatched, that call site is replaced by the body of the corresponding function, which typically allows for better optimisation.

To illustrate how the code generation is implemented we continue with the selection function as our running example:

```
int[d:shp]
sel(int[n] idx, int[n:outer,d:shp] arr)
{
    return { iv -> arr[idx ++ iv] | iv < shp };
}
```

In order to make use of the inlining capabilities of the compiler, we wrap this implementation function into a stub that we always allow to be inlined. This stub first checks the constraints of the given arguments with a new inline function `sel_pre`, and then applies the original function definition `sel_impl`. This replacement ensures that all applications of `sel` now point to this new function definition containing the corresponding checks, making it possible to execute this compiler step early in the compilation process, before function calls have been dispatched. Now if an application of `sel` is able to be dispatched, the generated checks can be inlined to the call site of this function application.

```
inline int[*]
sel(int[.] idx, int[*] arr)
{
    pred = sel_pre(idx, arr);
    result = sel_impl(idx, arr);
    return result;
}
```

Note that the arguments themselves no longer contain type patterns and are now SaC types, as these were transformed by ATP in section 4.1 in order to make these function signatures compatible with the type checker.

Similarly to [23], we need to ensure that the compiler does not optimise away the generated checks. To achieve this we define a new primitive function: `guard`, that acts as a barrier for optimisation and code re-writes. This function expects `n` arbitrary values, `n` corresponding types, and a boolean predicate. Each type at index i is the type of the corresponding value at index i as it has been defined by the function signature. The types of the result values of these guards are then based on these type arguments, instead of the actual types of these arguments. We require this to avoid premature optimisation resulting from a specialisation of the arguments. Then if the predicate is true this guard function behaves as an identity function on these `n` arguments, otherwise an error is raised and the program aborts. This behaviour can occur at run-time, but also at compile-time, allowing the compiler to optimise away checks that are statically known to be true.

We surround the application of `sel_impl` with guards on the input values and the result value, that rely on the predicate returned by `sel_pre`. This avoids premature optimisations based on either the input values or output type of `sel_impl`.

```
inline int[*]
sel(int[.] idx, int[*] arr)
{
    pred = sel_pre(idx, arr);
    idx, arr = guard(idx, arr,
                     int[.], int[*], pred);
    result = sel_impl(idx, arr);
    result = guard(result, int[*], pred);
    return result;
}
```

Since the implementation function might make use of the variables defined in the type patterns of its arguments, we prepend the generated assignment to the original function body. This allows the variable shp from the selection example to be used directly in the body of that function.

```
int[*]
sel_impl(int[.] idx, int[*] arr)
{
    n = shape(idx)[0];
    outer = take(n, shape(arr));
    d = dim(arr) - n;
    shp = take(d, drop(n, shape(arr)));

    return { iv -> arr[idx ++ iv] | iv < shp };
}
```

As opposed to the `sel` stub, this implementation function is not always inlined. We only inline this implementation function if the developer marked the original function as inline, which avoids us from inlining a function that the developer did not intend to.

The body of the check function `sel_pre` contains the generated assignments, followed by the generated checks. If all checks succeed true is returned, otherwise we abort the program with a descriptive error message. In order to comply with the type checker, these aborts are of the same type as `pred`. They are weaved into the program using if-expressions, which makes it easy to insert additional checks and makes for a simple and readable implementation.

```
inline bool
sel_pre(int[.] idx, int[*] arr)
{
    n = shape(idx)[0];
    outer = take(n, shape(arr));
    d = dim(arr) - n;
    shp = take(d, drop(n, shape(arr)));

    pred = true;
    pred = n == dim(arr) - d ? pred
        : abort("Type pattern error ...");
    return pred;
}
```

A drawback of pulling the generated assignments and checks out of the original function definition and moving them to separate functions, is that we require two instances of the assignments to the features of type patterns, once in the check in `sel_pre`, and once in the implementation function `sel_impl`. This drawback is mitigated by the fact that performance-critical code is often inlined, in which case common sub-expression elimination can be applied to remove these duplicate definitions. Additionally, these functions might insert assignments for features that are not used in the function body, such as the variable `outer` from the example. This issue is easily resolved by applying dead code removal.

5.3 Type patterns for return types

Thus far we have only considered the code generation for type patterns of arguments, but as we have seen in our running selection example, it is also possible to define type patterns on return types.

```
int[d:shp]
sel(int[n] idx, int[n:outer,d:shp] arr)
{
    return { iv -> arr[idx ++ iv] | iv < shp };
}
```

To support this we extend the definition of `sel` that was given in section 5.2. As opposed to the pre-check function, we insert a post-check function after the result has been computed and check whether that result adheres to the given type pattern constraints. As before, we include a guard to avoid these checks from being prematurely optimised away.

```
inline int[*]
sel(int[.] idx, int[*] arr)
{
    pred = sel_pre(idx, arr);
    idx, arr = guard(idx, arr,
                     int[.], int[*], pred);
    result = sel_impl(idx, arr);
    result = guard(result, int[*], pred);
    pred' = sel_post(idx, arr, result);
    result = guard(result, int[*], pred');
    return result;
}
```

The post-check function behaves similarly to the pre-check function, however it additionally expects the results of the implementation and checks whether those values adhere to the constraints imposed by their type patterns. In the case of our selection example, this means that we insert checks to compare the rank and shape of the result against the features `d` and `shp` respectively.

```
inline bool
sel_post(int[.] idx, int[*] arr, int[*] result)
{
    n = shape(idx)[0];
    outer = take(n, shape(arr));
    d = dim(arr) - n;
    shp = take(d, drop(n, shape(arr)));

    pred = true;
    pred = d == dim(result) ? pred
        : abort("Type pattern error ...");
    pred = all(shp == shape(result)) ? pred
        : abort("Type pattern error ...");
    return pred;
}
```

The addition of return-type constraints requires us to determine whether a feature occurring in a return type should be converted to an assignment or to a check within the post-check function. In the case of our selection example, the features `d` and `shp` of the return type are both already defined by the argument `arr`, thus in the post-check function we generate two expressions to check whether the rank and shape of the result match `d` and `shp` respectively.

However, consider a case where the return type is `int[m,m]`. Because `m` has not been defined in any of the arguments of the function signature, we are required to first generate an assignment to `m` instead. Namely, we first need to generate `m = shape(result)[0]`. This variable can then be used for the comparison of the second feature, `m == shape(result)[1]`. This distinction allows us to define constraints within a single return type, or between multiple return types, that are not dependent on the arguments.

In order to be able to make this distinction, it is no longer sufficient to only keep track of a single list of assignments and checks. Instead, in the algorithm described in section 5.1 we make a distinction between assignments and checks that are generated for type patterns of arguments, and assignments and checks that are generated for type patterns of return types. All that is required is a

case distinction, based on whether the constraint originates from the feature of an argument, or the feature of a return type. For the sake of brevity we do not discuss the exact implementation.

6 ARBITRARY CONSTRAINTS

Type patterns alone are not yet sufficient for generating arbitrarily complex constraints on arguments and return values. To preserve the benefits provided by keeping these dependencies contained to the function signature, we extend the syntax of functions by allowing them to contain any number of arbitrary expressions that operate on arguments, or on features of type patterns. These conditions can be any built-in or user-defined boolean expression without side-effects.

In the case of the selection function, we can include two additional conditions to check whether the index vector is non-negative, and that each index in the index vector is less than the corresponding element in the shape of the array. Ensuring that the index is always in bounds.

```
int[d:shp]
sel(int[n] idx, int[n:outer,d:shp] a)
    | all(0 <= idx), all(idx < outer)
{
    return { iv -> a[idx ++ iv] | iv < shp };
}
```

Because of the way we interleave constraints into the pre- and post-check functions, inserting such checks into the program requires little additional effort. The only significant work that remains is to decide for each condition whether it is a pre-condition, or a post-condition. If all variables that appear in a condition are defined by arguments or type patterns of arguments in a function signature, then we add a case to the pre-check function for this condition. Otherwise, if at least one variable occurring in the condition is only defined in the type patterns of the return values, we add that condition to the post-check function instead.

As with the other checks, the condition will be inserted into the code with an if-expression and an as precise as possible error message that describes which condition failed. The pre-check function, as previously defined in section 5.2, can now easily be extended to include these two new conditions:

```
inline bool
sel_pre(int[.] idx, int[*] arr)
{
    n = shape(idx)[0];
    outer = take(n, shape(arr));
    d = dim(arr) - n;
    shp = take(d, drop(n, shape(arr)));

    pred = true;
    pred = n == dim(arr) - d ? pred
        : abort("Type pattern error ...");
    pred = all(0 <= idx) ? pred
        : abort("Type pattern error ...");
    pred = all(idx < outer) ? pred
        : abort("Type pattern error ...");
    return pred;
}
```

We similarly extend the post-check function, for any remaining conditions that operate on features of return types. In the case of our selection example, the post-check function remains unchanged.

Inserting the conditions into these functions allows them to potentially be statically analysed, just as the checks generated by the type patterns. Whilst we are not able to generate an error message as precise as for type patterns, we are still able to tell the programmer which of the given conditions failed.

With the addition of these feature, our type patterns now provide a complete implementation that is strong enough to support arbitrarily complex conditions, allowing constraints on arguments and return values to be defined entirely in the function signature. Fully separating such domain constraints from the actual logic of these functions.

7 CASE STUDY

In section 3.2 we have seen an implementation of the take function, which is used in practice in many array programs. Applications of this function are prone to out-of-bounds errors: the shape vector argument can have too many elements, or one of its elements can be greater than the corresponding extent in the array argument. Following is an example of this first case, where the array argument is one-dimensional but a two-element shape argument is provided.

```
arr = genarray ([10], 0f);
res = take ([7,9], arr);
```

Without the use of type patterns, this invalid application of take produces the following compile time error:

```
argument #1 should not exceed the shape of
argument #2 of "_drop_SxV_"; types found:
int{2} and int[1]
```

This error may come at a surprise as our program does not even make direct use of _drop_SxV_. To find out where this comes from, users can enable a stack trace. For our example, this yields:

```
./example.sac:6: error:
   -- in Array::sel( int[2], float[.])
./example.sac:12: error:
   -- in _MAIN::take( int[2]{7,9}, float[.])
./example.sac:18: error:
   -- in _MAIN::foo( float[.])
```

In this case, we can see that the failing application of _drop_SxV_ stems from a call to the selection function in the standard library (here: Array) which in turn was called by our version of take. However, it is not immediately clear that an erroneous call of take is the culprit, as we see the entire stack trace. In particular when dealing with long call sequences, a manual check where the unwanted function call happens can become very tedious.

In contrast, the type pattern described in this paper enable the compiler to directly point to the erroneous call. With type patterns, we get the following pre-condition compile time error:

```
Type pattern error in application of take:
dimensionality of argument `arr' is too small,
could not assign a non-negative value to `m'
in `m:inner'
```

Similarly, type patterns also improve error messages from function implementations that do not meet the co-domain constraints, even if the domain constraints are met. For example, let us assume a problem within the definition of take, where the upper bound incorrectly uses a $\leq$ instead of $<$, i.e., we have:

```
return { iv -> arr[iv] | iv <= shp };
```

Furthermore, let us assume we now call `take([10], arr)`. Without type patterns, we get an error from the out of bounds selection:

```
Scalar constraint `SACp_eat_43 (10) <
SACp_emal_7118__uprf_2476 (10)' violated
```

If we add type patterns to the definition of `take` we instead get a more descriptive post-condition error message, which makes it clear that there is a mistake in the definition of `take`:

```
Type pattern error in definition of take: the
found value of `shp' in `n:shp' of the return
value is not equal to the value of argument `shp'
```

We even obtain this error for examples such as `take([8], arr)` where this bug goes unnoticed in the variant without type patterns.

8 RELATED WORK

Dependent types. Previous work has investigated the effectiveness of dependent types in practise, in languages such as Agda, Idris, Cayenne, and Coq [2, 6–8]. Such dependently types languages require that a given program can be fully analysed statically. This requirement can lead to undecidability and non-terminating type checking, especially in the context of rank-polymorphic programs. This requirement of static analysis is often not feasible in practise, for example when reading input from a file, or for other I/O operations. These cases would require explicit assertions on program inputs, or might require additional proofs to be given by the developer to aid the type system with proving static correctness. This undermines our quest for readability and programming productivity. Instead, we implement a hybrid approach that does not enforce that all constraints can be resolved statically, by instead allowing some constraints to be checked dynamically at run-time. By doing so we lose the guarantee that any program that can be compiled is also total. But we avoid undecidability and non-termination by deferring the constraints that could not be resolved statically to run-time, where they can always be checked dynamically.

Another approach based on dependent types is to instead encode constraints as first-order formulas, and verify them in collaboration with a theorem prover. This is the method applied by Qube [34], by using the Yices theorem prover [10]. This approach rules out all programs with type errors, but also rejects some programs that would actually behave well at run-time. Conversely, our approach never rejects programs that will behave well at run-time, but might instead allow programs with type errors, which are then found using run-time checks.

In [36] a dependent type system to specify an energy semantics of a programming language is described. This system is extended in [37] with records and other data types. However, an array type is lacking because of lacking bounds in the syntax. Using type patterns, these energy semantics can be extended with arrays and used to derive upper bounds for energy consumption.

Constraint resolution. Other languages make use of a more restricted form of dependent types, such as Remora [31, 32]. Remora views type checking and type inference as a constraint resolution problem, as we have done for type pattern analysis. Similarly to type patterns, Remora allows multiple ranks to be defined by the same variable in order to impose constraints on these ranks. Like us, they have found that static analysis of such rank-polymorphic programs is infeasible without additional proofs from the developer, instead also opting for a partially static and partially dynamic approach. However, their approach functions only as a type-checking mechanism. Whereas type patterns additionally aim to aid developers in simplifying and debugging their function definitions by allowing them to define arbitrarily complex shape-constraints.

Much like Remora and our proposed solution, Futhark defines ‘size parameters’ [21, 22]. These size parameters allow developers to specify the ranks of array arguments and return values. Just like type patterns, these size parameters can then also be used in their corresponding function bodies. Unlike type patterns, this approach does not allow for rank-polymorphism and thus is not applicable to (variable-rank) shape slices. Having this restriction makes it possible for Futhark to allow for functions of a higher order, and additionally makes it possible to analyse the correctness of these size parameters fully statically. Size parameters cannot be constructed using compound expressions, and must be defined explicitly. Size bindings expand upon size parameters by lifting these two restrictions [3].

Bound analysis and checking. The importance of eliminating array bound checking has been acknowledged in previous work [15, 38] in the context of ML, using a restricted form of dependent types. Similarly to type patterns, these methods deal with shape checking [24]; a well-studied field that is concerned with the detection of shape errors without handling the data stored within. Whereas these approaches always allow for static analysis, in real-world applications they require developers to aid the type-checker with manual proofs. Similarly to our approach, these methods allow for a variable to be defined multiple times, declaring that both their values should be the same. Whereas we allow for arbitrary constraints to be defined for these variables, these approaches only allow for (in)equality comparisons in order to remain statically decidable.

Alternatively there are approaches that capture pre- and post-conditions imposed by developers [28], in order to drive an inference system for statically eliminating out of bounds checks. Instead of relying on annotations given by developers, [41] infers a constraint system by applying a set of automatically generated test cases. The results of which aid a verifier in determining type refinements in dependent types. A drawback of this approach is that it is computationally expensive, and can only discover program invariants in a restricted search space.

These bounds can also improve the analysis of memory consumption. [36] describes ResAna, a system that statically derives memory bounds and consumption. These type patterns, even if checked dynamically, would help derive better bounds.

Hybrid types. Previous work [14, 27] has found that dependent types with full static resolution are not always feasible, and that a hybrid approach might be preferred, particularly in the context of imperative languages. Hybrid type systems allow constraints to be defined for arguments and return values. They are, like type patterns, then potentially resolved dynamically. Examples of languages that implement a hybrid type system are Sage and Spec# [4, 19]. Additionally, the Deputy tool [1, 9] developed for C allows for the specification of relations between data elements. Similar tools have been developed for C, such as CCured and Cyclone [20, 26].

Whereas such languages allow constraints to be defined on variables of any type, the proposed type patterns can be applied only to

structures that have a, potentially user-defined, definition of rank and shape. This restriction allows us to better tailor our solution to arrays, which results in a simple syntax that allows the same variable to be used in multiple constraints, as well as directly in the function body. Additionally, this restriction allows us to provide more descriptive error messages.

Pattern matching. The concept of pattern matching is integral to our approach. As the shape and rank of arrays are values themselves, such an approach feels natural. Whereas pattern matching is normally applied to values within the function body, we propose a hybrid approach that makes this pattern a part of an argument's type in the function signature. Pattern matching on array elements is already possible in a large selection of other languages. Examples of these are: structural pattern matching in Python [25], (sub)slice patterns in Rust[3,4], and list patterns in C#[5]. Whereas we match on the shapes and ranks of arrays, these approaches match on array elements and do not allow for the definition of constraints through multiple definitions of the same identifier.

Previous work [11–13] investigates how pattern matching can effectively be applied to unfree data types. Unfree data types are types whose data has no canonical form, such as sets. For example, the forms $\{1, 2\}$, $\{2, 1\}$, and $\{1, 2, 1\}$ all denote the same set. Non-linear pattern matching allows one to pattern match on such unfree data types, independently of their internal representation. These approaches share similarities with type patterns, because they also separate the representation of a data type from its actual data elements.

Single Assignment C. Our work is based on related work in the context of the SaC compiler [18, 23, 33], which investigates how constraints on array shapes can be introduced. We expand upon this work by integrating such constraints into function signatures, creating a stronger connection between the function signature and the shapes of its arguments and return values, as well as allowing for the generation of descriptive and precise error messages. Additionally we investigate how these constraints can be inserted into the code optimally, potentially allowing for more static analysis and better optimisation.

9 CONCLUSIONS

We present a new notation for array types which enables the specification of dependent type signatures that supports rank-polymorphism and allows for arbitrarily complex conditions, while maintaining flexibility and a high level of code readability.

Our notation aims to simplify earlier related work, by allowing shape and rank reuse across constraints and function bodies, reducing code complexity. By using a hybrid approach we are able to support rank-polymorphic programs, whilst still allowing for as much static analysis as possible, only dynamically checking those constraints that could not be resolved statically. We provide generic rules and algorithms which make it possible to implement type patterns into different compilers, and have shown that this is possible by implementing type patterns in SaC, and by applying them to the SaC standard library. As it turns out, the use of type patterns within the standard library does not only increase code readability but it also reduces the source code size considerably as shape deconstructing functions usually can be avoided within function definitions. Additionally, the use of type patterns exposed a previously unknown bug, that was hidden by a multitude of `shape` and `dim` operations. Furthermore the use of type patterns in education at the Radboud University has increased the student's understanding of array programs, by enabling them to better reason about shape relations between arguments and return values. These applications show the feasibility and effectiveness of type patterns in practise.

Another benefit of type patterns is the insertion of more descriptive and precise error messages. Previously, without additional work from the programmer and convoluting the code with manual checks, the cause of an error could be hard to track down due to a disconnect between the producer and consumer of an erroneous argument. With type patterns we find errors immediately when a function is applied erroneously, potentially already at compile time. We provide descriptive error messages, aiding programmers in finding and fixing bugs faster.

Although we develop our ideas in the context of SaC they are transferable to other languages. In particular array languages such as Futhark most likely could directly incorporate these ideas. Since Futhark already supports size parameters within its type system, the type pattern approach might be a suitable vehicle to introduce rank-polymorphism to Futhark without having to implement a fully dependent type system.

Whereas we chose to implement type patterns in the context of array programming, the ideas and rules provided in this paper are applicable to any language with types that have some (user-defined) rank and shape-component. The rules and algorithms provided in this paper, whilst somewhat tailored to the context of SaC, are defined generically so that they can be modified for, and implemented into, other languages. Because we allow for user-defined overloads of the rank and shape functions in SaC, type patterns can be applied to any type that implements these definitions. For example, a user might define the rank of a string to be equal to the number of words, with the shape being the length of each word. Alternatively they might define the rank of a tree to be equal to the number of leaves, and the shape equal to the depth of each of those leaves, or they might define the rank to be the number of strongly connected components in a graph, with the shape being the size of each of those strongly connected components. This allows type patterns to be beneficial in contexts other than array programming.

Future work might investigate how the additional shape information provided by type patterns can be delegated throughout the program to allow for even further static analysis, and to potentially use this information for further optimisation. Additionally, one might investigate whether a type-checker can be modified to find more compile-time errors by considering the relation between type patterns of return types, and the type patterns of following applications of these return types. Potentially allowing for type errors based solely on function signatures.

[3]https://blog.rust-lang.org/2018/05/10/Rust-1.26.html#basic-slice-patterns
[4]https://blog.rust-lang.org/2020/03/12/Rust-1.42.html#subslice-patterns
[5]https://learn.microsoft.com/dotnet/csharp/language-reference/operators/patterns#list-patterns

ACKNOWLEDGMENTS

The authors thank Peter Achten, Robert Bernecky, Thomas Koopman, and the anonymous reviewers, for proofreading the paper and providing helpful suggestions to improve it.

REFERENCES

[1] Zachary Ryan Anderson. 2007. *Static analysis of C for hybrid type checking*. Technical Report. Tech. Rep. EECS-2007-1, UC Berkeley.

[2] Lennart Augustsson. 1998. Cayenne—a Language with Dependent Types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/289423.289451

[3] Lubin Bailly, Troels Henriksen, and Martin Elsman. 2023. Shape-Constrained Array Programming with Size-Dependent Types. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC 2023)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3609024.3609412

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–69.

[5] Robert Bernecky, Stephan Herhut, and Sven-Bodo Scholz. 2010. Symbiotic Expressions. In *Implementation and Application of Functional Languages*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–124.

[6] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78.

[7] Edwin C. Brady. 2011. IDRIS —: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV '11)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/1929529.1929536

[8] Adam Chlipala. 2010. An Introduction to Programming and Proving with Dependent Types in Coq. *Journal of Formalized Reasoning* 3, 2 (Jan. 2010), 1–93. https://doi.org/10.6092/issn.1972-5787/1978

[9] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 520–535.

[10] Bruno Dutertre and Leonardo De Moura. 2006. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2, 2 (2006), 1–2.

[11] Satoshi Egi. 2014. Non-Linear Pattern-Matching against Unfree Data Types with Lexical Scoping. *CoRR* abs/1407.0729 (2014). arXiv:1407.0729 http://arxiv.org/abs/1407.0729

[12] Satoshi Egi and Yuichi Nishiwaki. 2018. Non-linear Pattern Matching with Backtracking for Non-free Data Types. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 3–23.

[13] Martin Erwig. 1997. Active patterns. In *Implementation of Functional Languages*, Werner Kluge (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–40.

[14] Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/1111037.1111059

[15] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Association for Computing Machinery, 268–277. https://doi.org/10.1145/113445.113468

[16] Clemens Grelck. 2012. *Single Assignment C (SAC) High Productivity Meets High Performance*. Springer Berlin Heidelberg, Berlin, Heidelberg, 207–278. https://doi.org/10.1007/978-3-642-32096-5_5

[17] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC–A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming* 34, 4 (01 Aug 2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

[18] Clemens Grelck, Fangyong Tang, et al. 2014. Towards Hybrid Array Types in SaC. In *Software Engineering (Workshops) (CEUR Workshop Proceedings)*, Vol. 1129. CEUR-WS.org, 129–145. https://ceur-ws.org/Vol-1129/paper44.pdf

[19] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. 2006. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, Vol. 6. 93–104.

[20] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. 2005. Cyclone: A type-safe dialect of C. *C/C++ Users Journal* 23, 1 (2005), 112–139.

[21] Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3460944.3464310

[22] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[23] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. 2008. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 254–273.

[24] C Barry Jay and Milan Sekanina. 1996. *Shape checking of array programs*. Technical Report. Citeseer.

[25] Tobias Kohn, Guido van Rossum, Gary Brandt Bucher II, Talin, and Ivan Levkivskyi. 2020. Dynamic Pattern Matching with Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3426422.3426983

[26] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. Association for Computing Machinery, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

[27] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

[28] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. 2008. A Practical and Precise Inference and Specializer for Array Bound Checks Elimination. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1328408.1328434

[29] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of functional programming* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[30] Sven-Bodo Scholz and Artjoms Šinkarovs. 2021. Tensor Comprehensions in SaC. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 13 pages. https://doi.org/10.1145/3412932.3412947

[31] Justin Slepak, Panagiotis Manolios, and Olin Shivers. 2018. Rank Polymorphism Viewed as a Constraint Problem. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2018)*. Association for Computing Machinery, New York, NY, USA, 34–41. https://doi.org/10.1145/3219753.3219758

[32] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

[33] Fangyong Tang, Clemens Grelck, et al. 2012. User-defined shape constraints in SaC. In *DRAFT PROCEEDINGS OF THE 24TH SYMPOSIUM ON IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES (IFL 2012)*. 416–434.

[34] Kai Trojahner. 2011. QUBE-Array Programming with Dependent Types. *Institute of Software Engineering and Programming Languages of the University of Lübeck* (2011).

[35] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643–664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[36] Bernard van Gastel, Rody Kersten, and Marko C. J. D. van Eekelen. 2015. Using Dependent Types to Define Energy Augmented Semantics of Programs. In *Foundational and Practical Aspects of Resource Analysis - 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Marko C. J. D. van Eekelen and Ugo Dal Lago (Eds.), Vol. 9964. 20–39. https://doi.org/10.1007/978-3-319-46559-3_2

[37] Bernard van Gastel and Marko C. J. D. van Eekelen. 2017. Towards Practical, Precise and Parametric Energy Analysis of IT Controlled Systems. In *Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017 (EPTCS)*, Guillaume Bonfante and Georg Moser (Eds.), Vol. 248. 24–37. https://doi.org/10.4204/EPTCS.248.7

[38] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/

277650.277732

[39] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

[40] Christoph Zenger. 1997. Indexed types. *Theoretical Computer Science* 187, 1 (1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5

[41] He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Dependent Array Type Inference from Tests. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 412–430.