Type Patterns: Pattern Matching on Shape-Carrying Array Types

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

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

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

ABSTRACT

In this paper we present a notion for shape-carrying array types that enables the specification of fully-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 them to double as patterns for matching shapes and shape-components from arguments, making them directly available in the respective 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 hybrid type system of SaC, where we try to statically resolve as many constraints as possible by means of partial evaluation. 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, Rank-Polymorphism, Dependent Types, Type Constraints, Partial Evaluation, Shape Pattern

ACM Reference Format:
Jordy Aaldering, Bernard van Gastel, and Sven-Bodo Scholz. 2023. Type Patterns: Pattern Matching on Shape-Carrying Array Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 computational tasks. In this domain, understanding and controlling the shape and rank 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, but often to argument values as well. Element-selection is a prominent example for such a case.

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 optimisations. The value-dependent nature of these constraints in the context of array programming languages has led to several different type systems [1–5], that try to balance the trade-offs between readability of code, decidability of the type system, and expressiveness of the language. While most of these approaches prioritise decidability, in this paper, we take a slightly different approach. Rather than starting out from a type system, we try to 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 argument shape-components in function bodies through these variables. By handling such type variables like normal argument names, arbitrary constraints between argument domains and result co-domains can be conveniently expressed. Similar notations exist in prior type-based work, such as [3, 4], 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 our type patterns, a full implementation within the context of a type system for many practical examples would raise decidability issues. Several programs would require explicit assertions on program inputs, other programs would require additional proofs to help the type system 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 additional modelling which would undermine 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 [7, 8], or that can produce run-time errors, which aims to assist programmers in identifying in-compatibilities at the earliest possible stage. We chose the type system of SaC (Single assignment C) [9] as our target here, since it already builds on the ideas of partial evaluation. At that stage, our work builds on previous work [10, 11] which investigates how constraints on array shapes can be introduced in the context of SaC.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our contributions are:

- Type Patterns as an amalgamation of type specification and pattern matching in the context of functions on arrays.
- A formal mapping from type patterns into the 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.
- Several examples demonstrating the readability, expressiveness, and effectiveness of the proposed approach.

2 INTRODUCING TYPE PATTERNS

In the sequel, 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.

2.1 Syntax of type patterns

We start out with the syntax of type patterns, shown in figure 1.

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

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

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

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

Figure 1: EBNF for type pattern

It constitutes an extension of the notion of types from SaC, providing more specificational flexibility and introducing variables within these. 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 of its shape.

Single dimensions can either be denoted by a fixed number, the '`.`' symbol, or a variable. A fixed number requires the corresponding shape component to match that very number, the '`.`' symbol allows for an arbitrary extend. If a variable is being used, the extend found needs to match all other occurrences of that very variable within a given function definition.[1] For example, consider the type pattern `int[n,n]` which 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 pattern support features that capture entire slices of a given shape. These are captured by the rule `<multiple>` in figure 1. Again, we distinguish three cases depending on the length 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 length 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 length of the slice and the second one relates to the slice itself. For the length 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 shape vector against the variable `shp`. Note here that variables after the '`:`' symbol such as `shp` denote slices and therefore always represent vectors, even if they match no ranks or individual ranks. 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 one-element vector rather than a scalar value.

Finally, the pattern `<id> ':' <id>` enables a match of a variable-length slice of an array shape. Here, the first variable matches against the length of the slice, whereas the second variable matches against the slice itself. For example, the type 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 chained together to create complex type patterns. Consider, for example, a type pattern `int[n,5,d:shp] v`. It matches any shape of at least rank 2, whose extend in the second axis is five, and whose remaining axes are stored in `shp`. Some matching shapes are `[0,5]`, `[3,5]`, `[0,5,1,2,3]`, and `[7,5,1]`.

The variable `n` can also be reused in the third feature, with `[n,5,n:shp] v`, which now denotes arrays of at least rank 2 whose extend in the first axis determines the number of ranks that follow after the first two axes. Some examples of matching shapes are:

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

2.2 Using type patterns

Using these type patterns, we can now easily extract the shape and the rank of an argument, allowing for simple definitions of the `dim` and `shape` functions:

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

[1] Note here, that such type pattern variables cannot also be used for local variable definitions within the same function body.

```
int[d]
shape (int[d:shp] array) {
  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 add constraints between multiple arguments of a function. For example, shape-polymorphic addition of two floating point arrays can be defined 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, and they also specify that the result shape will be exactly the same as well.

An example for more intricate shape relationships between argument and result shapes is the function take:

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

We can see here that the result has the same rank as the second argument, where the first n elements of the result shape are defined by the values of the first argument. These type patterns demonstrate an interesting case: when looking at the type pattern of a 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 without further restrictions. Fortunately, these cases can be identified and rejected statically when translating type pattern into explicit constraint checking code.

3 TRANSFORMING TYPE PATTERNS INTO TYPES AND CONSTRAINTS

Following, we have a function that combines multiple variable-rank features that span across multiple arguments.

```
foo (int[n:shp1,m:shp2] a, int[n,+,m:shp2] b)
```

In this function we first find that n is equal to the first element in the shape of b. This value is then used for n:shp in a, after which it is possible to figure out what the rank of m:shp2 is, as it must equal the remainder of the rank of a. Following this we can now generate constraints for m:shp2 in b.

As can be seen in this example, we cannot simply resolve type patterns one-by-one from left to right. Instead we might need to switch between multiple arguments and features in order to be able to resolve all dependencies.

As a result, we consider type patterns as a constraint resolution problem. We do so by not only generating the relevant constraints for each feature, but also including the dependencies that need to be resolved, e.g. the values we need, before this constraint can be resolved. In this section we propose two functions to implement this constraint resolution problem.

3.1 Type pattern analysis

Type patterns are generally not compatible with pre-existing type checkers by default. For example, in the context of SaC compiler, there exists no type int[n,5:shp]. Therefore, in this section we provide a method for converting such type patterns to pre-existing types. In the case of the given example, int[n,5:shp], this function converts that type pattern to the type int[+], which is already understood by the SaC compiler as an array with a non-zero rank. This conversion ensures that no modifications of the type checker are necessary for the implementation of type patterns, which would otherwise be a difficult task.

We define a function ATP that applies this conversion from any type pattern to a type that is already understood by the compiler, and to additionally collect information that we require when generating constraints in section 3.2. This method essentially converts type patterns to the types that the programmer would have written previously, before the existence of type patterns.

This function requires four arguments:

(1) The type including the remaining features of the type pattern, that are yet to be checked.
(2) An integer fshp for the current number of fixed shapes.
(3) An integer fdim for the current number of fixed ranks.
(4) A list vdim containing all identifiers of the type pattern that have a variable rank.

This function traverses the features of the type pattern, and finally returns the pre-existing type, as well as the resulting fdim and vdim, as these two values will be required when generating constraints in section 3.2. Following are some examples:

$$\mathtt{ATP}(\mathtt{int}[5,3,7],\ 0,\ 0,\ [\,]) = (\mathtt{int}[5,3,7],\ 3,\ [\,])$$
$$\mathtt{ATP}(\mathtt{int}[5,n,7],\ 0,\ 0,\ [\,]) = (\mathtt{int}[.,.,.],\ 3,\ [\,])$$
$$\mathtt{ATP}(\mathtt{int}[2{:}shp],\ 0,\ 0,\ [\,]) = (\mathtt{int}[.,.],\ 2,\ [\,])$$
$$\mathtt{ATP}(\mathtt{int}[d{:}shp],\ 0,\ 0,\ [\,]) = (\mathtt{int}[*],\ 0,\ [\mathrm{d}])$$
$$\mathtt{ATP}(\mathtt{int}[8,d{:}shp],\ 0,\ 0,\ [\,]) = (\mathtt{int}[+],\ 1,\ [\mathrm{d}])$$
$$\mathtt{ATP}(\mathtt{int}[d,d{:}shp],\ 0,\ 0,\ [\,]) = (\mathtt{int}[+],\ 1,\ [\mathrm{d}])$$

After the function has been applied to all features; if vdim is not empty, then the type has an unknown rank that is at least of length fdim. Otherwise, we statically know that the rank is equal to fdim. Additionally, if the number of fixed shapes and fixed ranks is equal, the type pattern only contains constant integers, e.g. int[5,3,7], and thus the entire shape is statically known. This results in the following base-case:

$$\mathtt{ATP}(type[\,],\ \mathrm{fshp},\ \mathrm{fdim},\ \mathrm{vdim}) = (\mathrm{T},\ \mathrm{fdim},\ \mathrm{vdim})$$

where

$$\mathrm{T} = \begin{cases} \text{array of unknown rank} & \text{if vdim} \neq \emptyset \text{ and fdim} = 0 \\ \text{array of non-zero rank} & \text{if vdim} \neq \emptyset \text{ and fdim} > 0 \\ \text{array of known shape} & \text{if fdim} = \text{fshp} \\ \text{array of known rank} & \text{otherwise} \end{cases}$$

Note that here we assume that the type-checker makes a distinction between arrays of known shape and rank, but this case can easily be modified to support languages with type checkers that do not make this distinction.

Also note that many actual checks for the types happen later in the code generation part, after the rank and shape have been resolved. Compiler errors regarding mismatched dimensions and shapes can also be thrown there.

In the remainder of this section we define the recursive cases for the ATP function. Starting with the cases for dots and identifiers, all that is needed is to increment the number of fixed dimensions.

$$\begin{aligned}&\texttt{ATP}(type[\,.\,, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim}+1,\ \text{vdim})\end{aligned}$$

$$\begin{aligned}&\texttt{ATP}(type[id, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim}+1,\ \text{vdim})\end{aligned}$$

Similarly for numbers, but since this value describes the exact length of this dimension, we also increment the number of fixed shapes.

$$\begin{aligned}&\texttt{ATP}(type[\texttt{n}, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp}+1,\ \text{fdim}+1,\ \text{vdim})\end{aligned}$$

If instead we match on a shape with a constant length, we increase the number of fixed dimensions by that amount.

$$\begin{aligned}&\texttt{ATP}(type[\texttt{n}:shp, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim}+\texttt{n},\ \text{vdim})\end{aligned}$$

When this length is an identifier, we add that identifier to the list of variable dimensions instead.

$$\begin{aligned}&\texttt{ATP}(type[dim:shp, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim},\ \text{vdim} +\!\!+ [\text{dim}])\end{aligned}$$

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

$$\begin{aligned}&\texttt{ATP}(type[*, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim},\ \text{vdim} +\!\!+ [*])\end{aligned}$$

$$\begin{aligned}&\texttt{ATP}(type[+, ...rest],\ \text{fshp},\ \text{fdim},\ \text{vdim})\\ &= \texttt{ATP}(type[rest],\ \text{fshp},\ \text{fdim}+1,\ \text{vdim} +\!\!+ [*])\end{aligned}$$

3.2 Constraint generation

We can now define a function GTC that traverses a type pattern and generates constraints for the features of that type pattern.

This function expects four arguments:

(1) The argument and type, including the remaining features of the type pattern that are yet to be checked.
(2) The current index in the shape vector of the argument.
(3) A set deps that contains the identifiers that thus far require to be defined to resolve the current feature.
(4) An environment σ that maps identifiers to lists of (dependency, constraint) tuples. These tuples describe the constraints for that identifier, and the dependencies that are required to have been resolved for this constraint.

The function is applied recursively to all features of the given type pattern. It then returns an environment that maps each feature to a corresponding set of (dependencies, constraint) tuples. In other words, this set contains all constraints, with corresponding dependencies, that must hold for that feature.

Consider a type pattern `int[x,x]`, we apply the function as follows: $\texttt{GTC}(int[x,x]\,v,\ 0,\ \emptyset,\ \emptyset) = \sigma$. The environment σ will then contain the entry x, which will be a lists with two elements. The first element contains the constraint that compares x against the first index in the shape vector, and the second element contains the constraint that compares x to the second index in that same shape vector. When generating code in section 4.1, this first constraint becomes an assignment to the variable x, whereas the second constraint becomes a condition that checks whether that constraint has the same value as the previously defined x.

In certain scenarios, it is possible that the generated constraints cause an out-of-bounds selection. For example, if the user provides a value to a function that has a lower rank than is defined in the type pattern. To avoid generating erroneous code, we additionally add a constraint to ensure that the rank is large enough. If there are no variable dimensions we check whether the rank is exactly equal to the number of fixed ranks. Otherwise we check whether it has at least that many dimensions. This results in the following base-case:

$$\begin{aligned}&\texttt{GTC}(type[]\,v,\ \text{cdim},\ \text{deps},\ \sigma)\\ &= \begin{cases}\sigma[v \mapsto (\text{deps},\ \texttt{dim}(v) == v.\text{fdim})] & \text{if vdim} = \emptyset\\ \sigma[v \mapsto (\text{deps},\ \texttt{dim}(v) >= v.\text{fdim})] & \text{otherwise}\end{cases}\end{aligned}$$

We write $\sigma[v \mapsto (\text{deps},\ \texttt{c})]$ to denote that a new tuple is added to the constraints list of feature v. This constraint is described by a piece of code c, and requires the dependencies deps to be resolved first because the computation of c relies on the resulting values of those dependencies.

In the remainder of this section we define the recursive cases for the GTC function. When we encounter a dot we only need to increment the dimensionality expression for subsequent patterns. Since in the case of a dot we do not care about the result, no constraints need to be added to the environment.

$$\begin{aligned}&\texttt{GTC}(type[\,.\,, ...rest]\,v,\ \text{cdim},\ \text{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\,v,\ \text{cdim}+1,\ \text{deps},\ \sigma)\end{aligned}$$

When we encounter a constant integer n, we similarly increment the dimensionality expression for subsequent patterns. We also need to add a constraint to the environment for the argument v, with the current set of dependencies. This constraint checks whether the length of the dimension at the current index is equal to the given number.

$$\begin{aligned}&\texttt{GTC}(type[\texttt{n}, ...rest]\,v,\ \text{cdim},\ \text{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\,v,\ \text{cdim}+1,\ \text{deps},\\ &\sigma[v \mapsto (\text{deps},\ \texttt{n} == \texttt{shape}(v)[\text{cdim}])])\end{aligned}$$

Again we increment the dimensionality expression, if we encounter an identifier that described the extend of a single rank. We also add a constraint for this identifier id, with the current dependencies. This constraint gets the length of the dimension at the current index in the shape.

$$\begin{aligned}&\texttt{GTC}(type[id, ...rest]\,v,\ \text{cdim},\ \text{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\,v,\ \text{cdim}+1,\ \text{deps},\\ &\sigma[id \mapsto (\text{deps},\ \texttt{shape}(v)[\text{cdim}])])\end{aligned}$$

In the case of a shape of fixed length, given a constant integer n, we instead increase the dimensionality expression by this fixed amount. We also require a constraint for the shape identifier shp with the current dependencies. This constraint skips the first cdim dimensions from the shape, and then takes the following n elements.

We define a function droptake(cdim, n, v) that drops the first cdim elements from the shape of v, and then takes the next n elements of that shape. The exact implementation depends on the compiler and the primitive methods that are available, but will likely look somewhat like: take(n, drop(cdim, shape(v))).

$$\begin{aligned}&\texttt{GTC}(type[\texttt{n}:shp, ...rest]\, v,\ \texttt{cdim},\ \texttt{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\, v,\ \texttt{cdim} + \texttt{n},\ \texttt{deps},\\ &\sigma[shp \mapsto (\texttt{deps},\ \texttt{droptake}(\texttt{cdim},\ \texttt{n},\ v))])\end{aligned}$$

Similarly we have the case for a feature of variable-rank, id:shp. In this case we add a constraint for id to the environment, which we get by removing the current dimensionality from the total dimensionality. To get the constraint for shp we skip the first cdim dimensions from the shape, and then take the number of expressions given by the value of the identifier id. Since the result of id is required for this constraint, we add it to the list of dependencies for this constraint as well.

We define another function, dimcalc(v, id), which builds an expression that computes the dimensionality of v using the fdim and vdim found in section 3.1, excluding the identifier id. Again the implementation will depend on the compiler, but will look similar to: dim(v) − v.fdim − sum(v.vdim \ {id}).

$$\begin{aligned}&\texttt{GTC}(type[id:shp, ...rest]\, v,\ \texttt{cdim},\ \texttt{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\, v,\ \texttt{cdim} + id,\ \texttt{deps} \cap \{id\},\\ &\sigma[id \mapsto (\texttt{deps} \cap v.\texttt{vdim} \setminus \{id\},\ \texttt{dimcalc}(v,\ id))]\\ &[shp \mapsto (\texttt{deps} \cap \{id\},\ \texttt{droptake}(\texttt{cdim},\ id,\ v))])\end{aligned}$$

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 first increment the cdim before continuing.

$$\begin{aligned}&\texttt{GTC}(type[+, ...rest]\, v,\ \texttt{cdim},\ \texttt{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\, v,\ \texttt{cdim} + 1,\ \texttt{deps},\ \sigma)\end{aligned}$$

$$\begin{aligned}&\texttt{GTC}(type[*, ...rest]\, v,\ \texttt{cdim},\ \texttt{deps},\ \sigma)\\ &= \texttt{GTC}(type[rest]\, v,\ \texttt{cdim},\ \texttt{deps},\ \sigma)\end{aligned}$$

4 CONSTRAINT RESOLUTION

We take a partial evaluation based approach based on the idea of symbiotic expressions. [12] 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 of the program. Unlike symbiotic expressions, we propose that the inserted code is not necessarily removed after optimisation, in order to allow for run-time errors regarding type patterns for constraints that could not statically be resolved.

4.1 Resolution algorithm

Using the constraint environment σ as computed in section 3.2, we can now generate the necessary assignments and checks for functions with type patterns. The pseudo-code described in figure 2 will generate a list of assignment statements and check expressions in the correct order, based on the constraints and the corresponding dependencies. These two lists of generated assignments and checks will be inserted into the code in section 4.2.

We need two additional sets for this algorithm:

(1) A set defined, which contains all identifiers that have thus far been defined. Initially, these are only the arguments themselves.
(2) A set exist, which contains all identifiers that have remaining constraints. These are initially all identifiers that occur in the features of this function signature.

We repeat the algorithm until there a no more changes. When there were no more changes after any iteration, this means that every constraint has been resolved. Or that there are multiple constraints that all cannot be resolved, because they all depend on each other in some way.

```
changed = false

foreach feature in exist:
  all_dependencies_resolved = true

  foreach (deps, constraint) in env[feature]:
    if (deps / defined) is not empty:
      all_dependencies_resolved = false
      continue

    if feature is not in defined:
      generate assignment "feature = constraint"
      defined = defined + {feature}
    else:
      generate check "feature == constraint"

    remove this entry from env[feature]
    changed = true

  if all_dependencies_resolved:
    exist = exist / {feature}
```

Figure 2: Resolution algorithm pseudo-code

This algorithm iterates over all features in the exist set, which are the features that have remaining constraints. We then iterate over these remaining (dependencies, constraint) tuples. If this constraint has dependencies that are not yet defined, e.g. the intersection of these two sets is not empty, then this constraint still has unresolved dependencies, and we skip it in this iteration.

Next we have a case distinction, depending on whether the identifier of this feature has a definition yet. If not, we add this feature to the defined list and generate an assignment to this identifier with the result of the constraint. Otherwise there already exists

such a definition., and we instead generate a check to ensure that this definition is equal to the result of the constraint.

Finally, we remove this entry from this feature's list of constraints, and we set changed to true. After no more changes occur, if `exist` is empty, we were able to resolve all constraints. Otherwise, `exist` contains all constraints that could not be resolved, which can now be shown to the user in an error message.

4.2 Code generation

We can now insert the assignments and checks, that we generated in the section 4.1, into the pre-existing code. To do so we first require a new primitive function: `guard`. This function expects `n` arbitrary values, and a boolean predicate. If the predicate is true, this guard function behaves as an identity function on these `n` arguments. Otherwise, a type error is raised and the program aborts.

We require this function in order to to ensure that the compiler does not optimise away the generated checks, unless the boolean result is statically known. If the result is statically known to be true, then the guard function is equal to the identify function, and can thus be removed. Otherwise, if the result is statically known to be false, and a type error can already be given at compile time.

```
v1, .., vn = guard (v1, .., vn, predicate);
```

Special care needs to be taken to ensure that the generated code is inserted in the most optimal location. To provide the compiler with as much information as possible, the generated assignments and checks should preferably be inserted around the call-site of the corresponding function. This makes it more likely that the checks can statically be resolved, allowing for more type errors at compile time. But with, for example, overloaded functions. The compiler might not have been able to statically dispatch the function call to a specific overload. In this case we have no choice but to insert the generated code into the function body.

To account for both cases without duplicating code, we generate multiple functions that we allow to be inlined, by doing so the compiler can automatically inline these functions whenever possible. By keeping the original function definition separate, we ensure that this definition is only inlined if the programmer stated that it should be. If all instances of these generated functions have been dispatched and inlined, dead code removal will automatically be able to remove these definitions.

As a running example, we will consider a function `foo`:

```
int
foo (int[n,n,n] a, int[n,m] b) {
    return n + m;
}
```

We generate a function, `foo_pre`, that checks whether the shapes of the arguments match their type patterns. This function contains the generated assignments, followed by the generated checks. If all checks succeed, true is returned. Otherwise, since most type-checkers have some concept of a bottom type, we return a bottom type. This bottom type also contains an error message, so that a useful message can be given to the programmer if a check fails. Using a bottom type also allows us to use the same notation for both the static and dynamic analysis. Additionally, at compile time SaC is able to collect multiple bottom types, allowing us to show an error message for all checks that we statically know failed.

```
inline bool
foo_pre (int[] a, int[] b) {
    n = shape(a)[0];
    m = shape(b)[1];

    pred = true;
    pred = n == shape(a)[1]
           ? pred : _|_ ("error message");
    pred = n == shape(a)[2]
           ? pred : _|_ ("error message");
    pred = n == shape(b)[0]
           ? pred : _|_ ("error message");
    return pred;
}
```

Note here that the arguments themselves no longer contain type patterns, as these were removed by ATP in section 3.1 to make these function signatures compatible with the pre-existing type checker.

Next, we insert the generated assignments, along with the original function body, into a new function: `foo_impl`.

```
int
foo_impl (int[] a, int[] b) {
    n = shape(a)[0];
    m = shape(b)[1];
    return n + m;
}
```

Finally, we replace the original definition by a new function that applies the previously implemented functions.

```
inline int
foo (int[] a, int[] b) {
    pred = foo_pre (a, b);
    a, b = guard (a, b, pred);
    return foo_impl (a, b);
}
```

Note here that original definition of `foo` is actually renamed to `foo_impl`, and that we create a new instance of `foo`. This ensures that all applications of `foo` now point to the new definition that now contains the corresponding checks, which makes it possible to execute this compiler step before functions have been dispatched. Then, if dispatching happens, calls to `foo` will point to the newly generated function instead.

One drawback of this approach is that we require multiple instances of the assignments to the features `n` and `m`, once in the check in `foo_pre`, and once in the implementation in `foo_impl`. This drawback is mitigated by the fact that performance-critical code is often inlined, in which case the compiler will be able to figure out that these assignments are the same, and will remove the duplicate definitions.

4.3 Error messages

When a function is provided with invalid arguments, it is crucial to give a good, human readable, error message. Type patterns provide a clear benefit here. Consider matrix multiplication as an example, it requires that the first argument has a shape $N \times M$ and that the second argument has a shape $M \times P$. This can be easily expressed using type patterns: `matmul (int[n,m] a, int [m,p] b)`.

Now, if the user provides an invalid input to this function, a descriptive error message can be given which describes that the value of `m` in argument `b` is not equal to `m` in argument `a`. Without type patterns, and without additional effort from the developer to

provide proper errors given incorrect inputs, we would not be able to get such a descriptive error. Instead, likely an out-of-bounds error will occur, which does not make it clear what actually is the cause of the problem, and ultimately, how it can be resolved.

Excluding the actual values of the arguments, which we often only know at run-time, the error message can be generated entirely at compile time. This makes it possible to reuse the same error message for both compile-time and run-time errors. For example, an error message for our matrix multiplication example could look as follows: `Type pattern error in definition of 'matmul' (line:column): 1st feature 'n' in type pattern of argument 'b' does not match feature 'n' in argument 'a'.`

4.4 Type patterns for return types

Thus far we have only considered type patterns for function arguments, but they can also be applied to the return values of functions. We will consider a modified version of the example used in section 4.2, which has an additional return value with a type pattern.

```
int, int[n,p,p]
foo (int[n,n,n] a, int[n,m] b) {
    return (n + m, a + 5);
}
```

We can use the same GTC rules as described previously in section 3.2. But since return values do not have a corresponding identifier in the function signature like arguments do, we need to ensure that each return value has a corresponding variable.

This means that a bit more work is required for some functions to make then compatible. Since the return value of `foo` has a type pattern, we need to store the results of `foo` in variables, so that we can apply the generated checks to those variables. Namely, we need to flatten return statements, which results in following function.

```
int, int[n,p,p]
foo (int[n,n,n] a, int[n,m] b) {
    x = n + m;
    y = a + 5;
    return (x, y);
}
```

By converting the code to this format, the required checks can now be inserted right before the return statement, making use of these newly generated variable names, `x` and `y`.

Features that are only defined in type patterns of return values can obviously not be used in the function body, as they depend on the result. For example, the identifier `p` defined in the example occurs only in the type pattern of the return value, and never in the argument's type patterns. Thus, only in the post-check function do we generate an assignment for `p` that depends on the returned value. Conversely, the identifier `n` is used in the type patterns of both the arguments and the return values, thus in this case we only generate a check for `n` in the post-check function that compares this value to the returned value.

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. 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 values.

We define another check function, similar to the one shown in section 4.2. This post-check function instead checks the features of the returned values.

```
inline bool
foo_post (int[] a, int[] b, int x, int[] y) {
    n = shape(a)[0];
    p = shape(y)[1];

    pred = true;
    pred = n == shape(y)[0]
           ? true : _|_ ("error message");
    pred = p == shape(y)[2]
           ? pred : _|_ ("error message");
    return pred;
}
```

Additionally, we need to modify `foo` to include this post-check function, along with an accompanying guard. This results in a modified definition of `foo`:

```
inline int, int[]
foo (int[] a, int[] b) {
    pred = foo_pre (a, b);
    a, b = guard (a, b, pred);
    x, y = foo_impl (a, b);
    pred = foo_post (a, b, x, y);
    x, y = guard (x, y, pred);
    return (x, y);
}
```

5 ARBITRARY CONSTRAINTS

These type patterns are not yet sufficient for generating arbitrarily complex conditions. To keep the benefits provided by containing array type dependencies to the function signature, we extend the syntax of functions by allowing them to contain arbitrary expressions that operate on features of type patterns. These expressions must be applications of user-defined boolean functions, or primitive functions and operators that return a boolean result.

```
int, int[n,p,p]
foo (int[n,n,n] a, int[n,m] b)
    | n > m, bar(m), p > 1, p % m == 2 {
    return (n + m, a + 5);
}
```

Because of our previous efforts, inserting these checks in the code is quite easy. The only additional work that needs to be done is to decide whether each condition should be inserted into the pre- or post-check function. If all features that appear in a condition are defined in the arguments of a function, then we add a case to the pre-check function. In our example, that would be the expressions '`n > m`' and '`bar(m)`'.

Otherwise, if at least one of the features occurring in the condition is only defined in the type pattern of a return value, we add the condition to the post-check function instead. In our example, that would be the expressions '`p > 1`' and '`p % m == 2`'.

As with the other checks, the condition will be inserted into the code with an if-expression, and a bottom type with some error message containing the condition that failed. The pre-check function, as previously defined in section 4.2, can now easily be extended to also include the arbitrarily complex conditions. This results in the following pre-check function:

```
bool
foo_pre (int[] a, int[] b) {
    n = shape(a)[0];
    m = shape(b)[1];

    pred = true;
    ...
    pred = n > m  ? pred : _|_ ("n > m failed");
    pred = bar(m) ? pred : _|_ ("bar(m) failed");
    return pred;
}
```

We similarly modify the post-check function, for expressions that contain features that are defined by at least one return value. This notation allows the conditions 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 this feature, our type patterns are now strong enough to support arbitrarily complex conditions, allowing constraints on arguments and return values to be defined entirely in the function signature.

6 RELATED WORK

Previous work has investigated the effectiveness of dependent types in practise, in languages such as Agda, Idris, and Coq. [13–15] These languages require that a given program can be fully analysed statically, e.g. that the program must be total. This requirement that programs must be total can lead to undecidability and non-terminating type checking, especially in the context of shapes and shape-components, and for rank-polymorphic programs. Additionally, this requirement of static analysis is often not feasible, for example when reading input from a file, and for other I/O operations. We implement a hybrid approach, that does not enforce that all constraints can be resolved statically, by instead inserting the required constraints into function bodies and allowing them to potentially be analyses dynamically. 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 we could not resolve statically to run-time, where they can be analysed dynamically.

Similarly there is the dependently typed language Qube. [16] Qube considers the relation between types and type systems, and applies the Yices theorem prover for type checking, in order to sidestep the undecidability issues in the previously discussed approaches. 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 will never reject programs that will behave well at run-time, but might instead allow programs with type errors. These type errors can then only be found at run-time.

A language that does not rely on dependent types is Futhark. [17] Similarly to our proposed solution, Futhark has 'size parameters'. [4] These size parameters allow programmers to specify that certain ranks of one or more array arguments must have a certain length. Just like our proposed solution, Futhark allows these size parameters to be used in the function body. Unlike type patterns, this approach is not rank-polymorphic. Having this restriction makes it possible to allow for functions of a higher order in Futhark, and additionally makes it possible to analyse the correctness of these size parameters statically.

A different approach, that also does not rely on static resolution, are hybrid type systems. [18] Similarly to our approach, hybrid type systems allow constraints to be defined for arguments and return values, which are potentially resolved dynamically. Examples of languages that implement a hybrid type checker are Sage and Spec#. [19, 20] Whereas such languages allow constraints to be defined for variables of any type, we only support constraints on structures with a length-component, which in the case of SaC are always arrays. This restriction allows us to better tailor our solution to arrays, which resulted in a simple syntax that additionally allows shape-components to be used in other constraints, and directly in the function body. Additionally it allows us to provide more descriptive error messages.

Our work is based on related work in the context of the SaC compiler, which investigates how constraints on array shapes can be introduced. [10, 11] We expand upon this work by integrating such constraints into function signatures, creating a strong link between the function signature and the shapes of its arguments, as well as allowing for the generation of descriptive error messages. Additionally we investigate how these constraints can be inserted into the code optimally, potentially allowing for more static analysis and optimisation.

The idea of pattern matching is integral to our approach. [21] As shapes 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.

7 CONCLUSION AND FUTURE WORK

In this paper we have presented a new notion for array types that enables the specification of fully-dependent type signatures that supports rank-polymorphism and allows for arbitrarily complex conditions, while maintaining flexibility and a high level of code readability. Our approach differentiates itself from related work by providing a simple notation that allows the re-use of shape-components in other constraints and the function body, reducing code complexity and strengthening the link between the function signature and its body. By using a hybrid approach we are able to allow for rank-polymorphism, whilst still allowing for as much static analysis as possible. Only dynamically checking those constraints that could not be resolved statically. We have provided generic rules and algorithms that 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. Showing the feasibility of type patterns in the real world.

The approach described in this paper can not only be applied to SaC, but can be beneficial to any language that has structures with length-components. The rules and algorithms provided in this paper, whilst somewhat tailored to work well with SaC, have been defined as generically as possible so that they can be modified and implemented into any other language. An example of such a language that can benefit from type patterns is Futhark. Whereas the pre-existing size parameters in Futhark can only be applied to

arguments of a known rank, by implementing type patterns Futhark might allow for rank-polymorphic programs without having to implement a dependent type system.

Another clear benefit of type patterns is the insertion of descriptive error messages. Previously, without additional work from the programmer, the cause of an error could be hard to track down. This happens because errors were only found as late as possible, for example when out-of-bound indexing occurs as a result of an invalid input. With type patterns, we find errors immediately when a function is applied erroneously and apply a descriptive error, allowing programmers to find and fix bugs faster.

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 only by investigating how arguments are constrained by being applied to functions with type patterns. Which would rely on the relation between the type patterns of arguments, and the type patterns return values.

REFERENCES

[1] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1):147–165, 1997.

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

[3] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

[4] Troels Henriksen and Martin Elsman. Towards size-dependent types for array programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–14, 2021.

[5] Justin Slepak, Panagiotis Manolios, and Olin Shivers. 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, page 34–41, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Artjoms Sinkarovs, Thomas Koopman, and Sven-Bodo Scholz. Rank-polymorphism for shape-guided blocking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '23)*. ACM, 2023.

[7] Robert Bernecky, Stephan Herhut, and Sven-Bodo Scholz. Symbiotic expressions. In Marco T. Morazan and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages, 21st International Symposium, IFL 2009, South Orange, NJ, USA*, number 6041 in Lecture Notes in Computer Science, pages 107–126. Springer, 2010.

[8] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. From contracts towards dependent types: Proofs by partial evaluation. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19*, pages 254–273. Springer, 2008.

[9] Clemens Grelck and Sven-Bodo Scholz. Sac—a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34:383–427, 2006.

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

[11] Clemens Grelck, Fangyong Tang, et al. Towards hybrid array types in sac. In *Software Engineering (Workshops)*, pages 129–145, 2014.

[12] Robert Bernecky, Stephan Herhut, and Sven-Bodo Scholz. Symbiotic expressions. In *International Symposium on Implementation and Application of Functional Languages*, pages 107–124. Springer, 2009.

[13] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 73–78. Springer, 2009.

[14] Edwin C Brady. Idris— systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54, 2011.

[15] Adam Chlipala. An introduction to programming and proving with dependent types in coq. *Journal of Formalized Reasoning*, 3(2):1–93, 2010.

[16] Kai Trojahner. Qube–array programming with dependent types. *Institute of Software Engineering and Programming Languages of the University of Lübeck*, 2011.

[17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 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*, pages 556–571, 2017.

[18] Cormac Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, 2006.

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

[20] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[21] Thierry Coquand. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, volume 92, pages 66–79. Citeseer, 1992.