On Optimising Shape-Generic Array Programs Using Symbolic Structural Information

Kai Trojahner[1], Clemens Grelck[2], and Sven-Bodo Scholz[2]

[1] University of Lübeck
Institute of Software Technology and Programming Languages
`trojahner@isp.uni-luebeck.de`
[2] University of Hertfordshire
Department of Computer Science
{`c.grelck,s.scholz`}`@herts.ac.uk`

Abstract. Shape-generic programming and high run time performance do match if generic source code is systematically specialised into non-generic executable code. However, as soon as we drop the assumption of whole-world knowledge or refrain from specialisation for other reasons, compiled generic code is substantially less efficient. Limited effectiveness of code optimisation techniques due to the inherent lack of knowledge about the structural properties of arrays can be identified as the single most important source of inefficiency.

However, in many cases partial structural information or structural relationships between arrays would actually suffice for optimisation. We propose symbolic array attributes as a uniform scheme to infer and to represent partial and relational structural information in shape-generic array code. By reusing the regular language to express structural properties in intermediate code, existing optimisations benefit from symbolic array attributes with little or no alteration. In fact, program optimisation and identification of structural properties cross-fertilise each other. We outline our approach in the context of the functional array language SaC and demonstrate its effectiveness by a small case study.

1 Introduction

Shape-generic array programming means writing functions, modules and entire programs in a style that completely or at least partially abstracts from the structural properties of the arrays involved. For example, a shape-generic matrix multiplication function is one that is applicable to pairs of matrices of any size, as long as the extent of the second axis of the first matrix equals the extent of the first axis of the second matrix. In fact, shape-generic array programming even goes one step further and allows functions to abstract not only from the extents of arrays along given axes, but even from the number of axes (or dimensionality or rank). For example, the element-wise multiplication of two arrays can be specified exactly once and applied to pairs of vectors, matrices, tensors and even higher-dimensional arrays.

The functional array programming language SaC [1,2] supports shape-generic array programming. Multi-dimensional arrays are characterised by their *rank*, an integer denoting the number of axes of an array, and their *shape*, a vector containing the extent along each axis. By step-wise abstraction from rank and shape, the type system of SaC distinguishes between three classes of array types, named *shape classes*:

1. Array of Known Shape (AKS),
2. Array of Known Dimensionality (AKD) and
3. Array of Unknown Dimensionality (AUD).

An AKS type, for example `int[10,10]`, describes the set of all arrays of some base type and a certain shape (including fixed rank). An AKD type, for example `int[.,.]`, defines the number of axes, but leaves the extent along each axis open. Finally, an AUD type like `int[*]` encompasses all arrays of a given base type regardless of their structure. Together, the array types form a natural hierarchy that induces a subtype relationship, e.g. `int[10,10]` $\prec$ `int[.,.]` $\prec$ `int[*]`.

When it comes to compiling shape-generic programs into executable code, it turns out that the run time performance of compiled AUD code is significantly inferior to compiled AKD code, which in turn is significantly inferior to compiled AKS (i.e. non-generic) code (cf. Section 5). This observation can be attributed to essentially two independent sources of inefficiency: Firstly, generic code requires the shape vector to be maintained in the heap at run time rather than being a set of compile time constants. The lack of a static rank knowledge also entails that no suitable nesting of loops can be generated to traverse an array. Instead, we must rely on a relatively inefficient loop structure. Secondly, and more gravely, many optimisation techniques are less effective if they lack structural information on the arrays involved. This holds for standard code transformations like constant folding, common subexpression elimination and loop unrolling just as for array-specific optimisations [3,4,5] or optimisations in the memory management subsystem [6].

Our recent work focussed on careful identification of where and how far to specialise [7]. Still, specialisation is not always the solution since sufficient structural information may lack at compile time or the number of specialisations may grow beyond feasability. Fortunately, many code optimisation techniques can benefit from more modest gains in structural knowledge than those resulting from specialisation. For example, it may be useful to know the extent of an AKD array along certain but not all axes. Or, we may exploit that an AUD array has at least two inner axes of which we may even figure out the extent. Or, we may improve programs utilising the knowledge that certain arrays have the same shape or the same dimensionality as others.

This work aims at making such fine-grained structural information below the level of shape classes available to optimisations by compile time inference. We introduce *symbolic array attributes* as uniform representations of the ranks and shapes of all arrays involved in an intermediate code representation. More precisely, we associate each definition of an array with new symbolic identifiers for its rank and its shape. Like an array identifier is bound to an expression defining that array, the associated rank and shape identifiers are bound to expressions that extract the definition of the array's rank and shape from its original definition. Any array-valued expression effectively is replaced by a triple of expressions. The first one is the original expression defining both structure and element values of the array. The second expression defines the shape, but abstracts from element values. The third expression defines the rank, but abstracts from concrete extents along the array's axes. Whereas the original code describes the relationships between arrays on the level of rank, shape and element values all at the same time, the augmented code explicates these relationships at each level individually.

Since all three elements of the expression triples are regular expressions of the language, they are automatically subject to a plethora of optimisations like constant folding, constant/variable propagation or common subexpression elimination to name just a few. This has a dual effect: The optimisations benefit from symbolic structural information while at the same time the structural information is improved by the optimisations.

It is characteristic for our approach to represent and exploit structural and relational properties of arrays in a purely compiler directed way. In particular, the source language remains entirely unaffected. An alternative approach would be to refine the type system towards using a variant of dependent types [8]. In contrast to our work, that would require a substantial extension to SaC as a programming language having a major impact on the style of programming.

The remainder of this paper is organised as follows: Section 2 defines a core language that we use to illustrate our ideas. We formally define how we introduce symbolic array attributes in Section 3 and how we use them for optimisation in Section 4. In Section 5 we illustrate our approach by a small case study. We discuss some related work in Section 6 and conclude in Section 7.

2 Introducing SaC$_{\mathsf{mini}}$

Many features of SaC are irrelevant for the context of this paper. Therefore, we define a core language called SaC$_{\mathsf{mini}}$, which exposes the relevant features in a condensed and simplified form. As defined in Fig. 1, a SaC$_{\mathsf{mini}}$ program is a sequence of potentially mutually recursive function definitions. Each function definition consists of a return type, a function name, a typed parameter list in parentheses and a code block. A code block is a sequence of variable-expression bindings terminated by a goal-expression, that follows the key word `return`. Alternatively, we may have a conditional where each branch either leads to a further conditional or terminates with a goal-expression. This SaC-style notation merely is a syntactic variation of a nesting of let-expressions and conditional expressions in other functional languages.

Any expression (and hence any variable) denotes an array, which is characterised by a *rank scalar*, a *shape vector* and a *data vector*. While the latter acts as a store for element values, all structural information is encoded in the rank scalar and the shape vector. The rank scalar describes the rank or dimensionality

```
Program     ⇒ [FunDef]*
FunDef      ⇒ Type Id ( [ Param [ , Param ]* ] ) Block
Param       ⇒ Type Id
Block       ⇒ { [Id = Expr ; ]* Return }
            | { [Id = Expr ; ]* Cond }
Return      ⇒ return ( Id ) ;
Cond        ⇒ if ( Id ) Block else Block
Expr        ⇒ Const | Id | FunAp | Vector | With
FunAp       ⇒ Fun ( [ Id [ , Id ]* ] )
Vector      ⇒ [ [ Id [ , Id ]* ] ]
With        ⇒ with [Generator Block ]* genarray ( Id , Id )
Generator   ⇒ ( [Id <= ] Id < Id )
Type        ⇒ AKS-Type | AKD-Type | AUD-Type
AKS-Type    ⇒ BaseType [ [ IntConst [ , IntConst ]* ] ]
AKD-Type    ⇒ BaseType [ . [ , . ]* ]
AUD-Type    ⇒ BaseType [ * ]
```

Fig. 1. Syntax of SAC$_{\mathsf{mini}}$

of the array; the shape vector describes an array's extent along each dimension. Consequently, the rank scalar denotes the length of the shape vector. In this model, scalars are rank zero arrays with an empty shape vector and a data vector consisting of a single element. Arrays can be nested as long as the whole array remains representable by rank, shape and data vector, i.e., all elements of an array must have the same element type and shape.

In addition to the usual scalar constants and identifiers, expressions may be applications of defined or of built-in functions. Built-in functions include the usual arithmetic, logic and relational operators on scalars. Whenever appropriate, we use infix notation for applications to improve readability. Array-specific built-in operations are limited to the following:

– `dim(`*`A`*`)` yields the rank scalar of array *`A`*.
– `shape(`*`A`*`)` yields the shape vector of array *`A`*.
– `sel(`*`iv`*`,`*`A`*`)` yields the element of *`A`* at the index specified by the integer vector *`iv`*.
– `modarray(`*`A`*`,`*`iv`*`,`*`v`*`)` yields a new array that is equivalent to *`A`* except for the element at index position *`iv`*, which is set to the scalar value *`v`*.
– `reshape(`*`shp`*`,`*`A`*`)` returns a new array whose data vector is given by the one of *`A`* but whose shape vector equals *`shp`*.

In applications of both `sel` and `modarray` the length of the index vector must coincide with the rank of the array; in applications of `reshape` the product of the elements of the desired shape vector must match that of the elements of the existing shape vector.

Unlike SaC, SaC$_{\mathsf{mini}}$ only supports non-nested expressions, i.e., arguments to a function application for example may only be identifiers, but not expressions again. This restriction simplifies the definition of compilation schemes; it can easily be achieved in a preprocessing step (from full SaC) by recursively extracting nested expressions and binding them to new identifiers. Nevertheless, we allow ourselves to use nested expressions wherever appropriate to improve the readability of code examples.

SaC$_{\mathsf{mini}}$ also features a simplified version of SaC's versatile array comprehension construct called WITH-loop. A WITH-loop of the form

`with ... genarray(`*`shp`*`,`*`def`*`)`

defines an array whose shape is given by appending the integer vector *shp* with the shape of the *default value* *def*. Each element of a WITH-loop-defined array is either set to the default value or computed according to the specification given in one of the *parts*. Each part consists of a *generator*, which defines a set of index positions, and an associated expression block, which determines the values of array elements at index positions covered by the generator.

A generator `(`*`lb`*`<=`*`iv`*`<`*`ub`*`)` defines a rectangular index range delimited by a lower bound vector *lb* and an upper bound vector *ub*. A missing lower bound specification defaults to a zero vector with the length of *ub*. The index variable *iv* is introduced in the generator, and its scope is limited to the associated expression; it represents the current index position. Multiple parts allow us to define different array elements according to different specifications. In order to ensure deterministic results, the index sets defined by the various generators of an individual WITH-loop must be pairwise disjoint.

SaC$_{\mathsf{mini}}$ and likewise SaC only have a very small set of built-in functions on arrays. A comprehensive set of compound operations on arrays is provided as a

```
bool select( int idx, bool[.] array)
{
  res = sel( [idx], array);
  return( res);
}

bool[*] select( int[.] idx, bool[*] array)
{
  shp = drop( select( 0, shape( idx)), shape( array));

  res = with (iv < shp) {
          elem = sel( idx ++ iv, array);
          return( elem);
        } genarray( shp, 0);
  return( res);
}
```

Fig. 2. Generalised selection functions

standard library, where they are defined by means of WITH-loops. Even many existing primitive functions are not intended for the general use, but rather serve as implementation vehicles for more general standard library functions. As an example, take the definition of a general selection facility in Fig. 2. The first instance of `select` takes a single integer and a vector.[1] In this case, the type information is sufficient to directly apply the built-in primitive `sel` without risking a run time error. The second instance of `select` implements the general case of selection: If the length of the selection vector is less than the dimensionality of the array to be selected from, selection yields an entire subarray. We achieve this by first dropping as many elements from the shape vector of the array as given by the length of the selection vector before we create an array of that shape. In the most relevant special case, the length of the selection vector actually coincides with the dimensionality of the array such that the application of `drop` yields the empty vector. Hence, the subsequent WITH-loop creates an array with an empty shape vector, which effectively is a scalar. Both auxiliary functions `drop` and vector concatenation `(++)` can be found in Fig. 3. For a more detailed explanation of the various SAC language features see [2]; a formal semantics may be found in [7].

```
int[.] drop( int v, int[.] a)
{
  dl = shape(a)[0] - v;
  ds = [dl];

  res = with ([i] < ds) {
            drel = a[i + v];
            return( drel);
        } genarray( ds, 0);
  return( res);
}

int[.] (++) (int[.] a, int[.] b)
{
  sa = shape(a);
  sb = shape(b);

  res = with ([i] < sa) {
          ael = a[i];
          return( ael);
        }
        ( sa <= [i] < sa + sb) {
          bel = b[i - sa[0]];
          return( bel);
        } genarray( sa + sb, 0);
  return( res);
}
```

Fig. 3. Auxiliary functions needed for the generalised selection

[1] We use the base type `bool` here as an example only.

3 Symbolic Array Attributes

In non-generic array code (shape class AKS) any structural relationship between arrays is properly expressed by their types. In non-specialised generic code, however, this property is immediately lost. For example, an application of the built-in function `modarray`

```
v = modarray(a,iv,0);
```

is known to yield an array `v` with a shape identical to that of the first argument `a`. Hence, the type inference system assigns `v` the type of `a`. Supposed `a` has a non-generic AKS type, this accurately reflects the structural relationship between `a` and `v`. However, if `a` has a generic AKD type say `int[.]`, then `v` is also assigned the type `int[.]`. This still reflects that `a` and `v` do have the same rank, but the fact that both actually have the same shape is not expressed. In the AUD case, we do not even know the equality of rank. This lack of information severely limits our opportunities for code optimisation.

Symbolic array attributes are meant to fill this gap and provide a systematic means to express partial structural information both with respect to individual arrays as well as structural relationships between different arrays. We augment any variable-expression binding in a function body with two (flattened) expressions: one to denote the array's rank (enclosed in round brackets) and one to denote the array's shape (enclosed in square brackets):

$$(v_d)\,[v_s]\ \ v\ =\ expr;$$

Only scalar constants, constant arrays and identifiers may occur in attribute positions. More complex sub-expressions are lifted into additional variable-expression bindings. Thus, despite appearing on the left-hand side of the assignment operator, symbolic array attributes are no less proper expressions than those on the right-hand side. Depending on the shape class of an array, the contents of v_d and v_s may vary, as outlined in the table below.

Shape class	v_d	v_s	Example
AKS	*Const*	*Array const*	`(0)[[]]`
			`(2)[[10,10]]`
AKD	*Const*	*Id*	`(2)[s]`
AUD	*Id*	*Id*	`(d)[s]`

Although rank and shape of an array may not be known until run time, we can consult their symbolic compile time representations using the attribute access functions $\mathcal{D}$ and $\mathcal{S}$. If the identifier *a* has been attributed with the pair `(d)[s]`, then $\mathcal{D}(a)$ gives *d* and $\mathcal{S}(a)$ yields *s*. The knowledge about the shape-preserving properties of `modarray`, can now be encoded by assigning the result *v* exactly those attributes of the modified array *a*:

$$(\mathcal{D}(a))\,[\mathcal{S}(a)]\ \ v\ =\ \texttt{modarray}(a, iv, val);$$

In Fig. 4 we show the transformation scheme $\mathcal{SAA}$ that introduces symbolic array attributes and, thus, makes array ranks and shapes explicit in terms of SaC$_{\mathsf{mini}}$ expressions. Rules of the form

$$\mathcal{C}\,[\![\, expr \,]\!] = expr\prime$$

$$\mathcal{SAA}\llbracket\, \mathit{type}\ \mathit{fun}(\mathit{params})\ \{\ \mathit{body}\ \}\ \rrbracket$$
$$= \mathit{type}\ \mathit{fun}(\mathcal{R}\llbracket \mathit{params} \rrbracket)\ \{\ \mathcal{MIR}\llbracket \mathit{params} \rrbracket;\ \mathcal{SAA}\llbracket \mathit{body} \rrbracket\ \}$$

$$\mathcal{SAA}\llbracket \texttt{if}\ (c)\ \mathit{then}\ \texttt{else}\ \mathit{else} \rrbracket = \texttt{if}\ (c)\ \mathcal{SAA}\llbracket \mathit{then} \rrbracket\ \texttt{else}\ \mathcal{SAA}\llbracket \mathit{else} \rrbracket$$

$$\mathcal{SAA}\llbracket \texttt{return}(a); \rrbracket = \texttt{return}(a);$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ c;\ R \rrbracket = \texttt{(0)[[]]}\ v\ \texttt{=}\ c;\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \mathit{sclprf}(\mathit{args});\ R \rrbracket = \texttt{(0)[[]]}\ v\ \texttt{=}\ \mathit{sclprf}(\mathit{args});\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ a;\ R \rrbracket = (\mathcal{D}(a))[\mathcal{S}(a)]\ v\ \texttt{=}\ a;\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \texttt{shape}(a);\ R \rrbracket = \begin{cases} \texttt{(1)[[1]]}\ v_s\ \texttt{=}\ [\mathcal{D}(a)]; \\ \texttt{(1)}[v_s]\quad v\ \texttt{=}\ \texttt{shape}(a); \\ \mathcal{SAA}\llbracket R \rrbracket \end{cases}$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \texttt{reshape}(s,a);\ R \rrbracket = \begin{cases} \texttt{(0)[[]]}\quad v_d\ \texttt{=}\ \mathcal{S}(s)\texttt{[0]}; \\ (v_d)[s]\quad v\ \texttt{=}\ \texttt{reshape}(s,a); \\ \mathcal{SAA}\llbracket R \rrbracket \end{cases}$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \texttt{modarray}(a,iv,v);\ R \rrbracket$$
$$= (\mathcal{D}(a))[\mathcal{S}(a)]\ v\ \texttt{=}\ \texttt{modarray}(a,iv,v);\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \texttt{[]};\ R \rrbracket = \texttt{(1)[[0]] v = [];}\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ [a_0,\ldots,a_{n-1}];\ R \rrbracket = \begin{cases} \texttt{(0)[[]]}\quad v_d\ \texttt{=}\ \texttt{1 +}\ \mathcal{D}(a_0); \\ \texttt{(1)[[1]]}\ v_{s_s}\ \texttt{=}\ [v_d]; \\ \texttt{(1)}[v_{s_s}]\ v_s\ \texttt{=}\ [n]\ \texttt{++}\ \mathcal{S}(a_0); \\ (v_d)[v_s]\ v\ \texttt{=}\ [a_0,\ldots,a_{n-1}]; \\ \mathcal{SAA}\llbracket R \rrbracket \end{cases}$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \mathit{fun}(\mathit{args});\ R \rrbracket = \mathcal{R}\llbracket v \rrbracket \texttt{=}\ \mathit{fun}(\mathit{args}); \mathcal{MIR}\llbracket \mathcal{T}\llbracket \mathit{fun} \rrbracket v \rrbracket; \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}\llbracket v\ \texttt{=}\ \texttt{with}\ \mathit{parts}\ \texttt{genarray}(s,d);\ R \rrbracket$$
$$= \begin{cases} \texttt{(0)[[]]}\quad v_d\ \texttt{=}\ \mathcal{S}(s)\texttt{[0]}; \\ (v_d)[s]\quad v\ \texttt{=}\ \texttt{with}\ \mathcal{SAA}\llbracket \mathit{parts} \rrbracket\ \texttt{genarray}(s,d); \\ \mathcal{SAA}\llbracket R \rrbracket \end{cases} \Bigg|\ \text{if D(d)} \equiv 0$$
$$= \begin{cases} \texttt{(0)[[]]}\quad s_{s_0}\ \texttt{=}\ \mathcal{S}(s)\texttt{[0]}; \\ \texttt{(0)[[]]}\quad v_d\ \texttt{=}\ s_{s_0}\ \texttt{+}\ \mathcal{D}(d); \\ \texttt{(1)[[1]]}\ v_{s_s}\ \texttt{=}\ [v_d]; \\ \texttt{(1)}[v_{s_s}]\ v_s\ \texttt{=}\ s\ \texttt{++}\ \mathcal{S}(d); \\ (v_d)[v_s]\ v\ \texttt{=}\ \texttt{with}\ \mathcal{SAA}\llbracket \mathit{parts} \rrbracket\ \texttt{genarray}(s,d); \\ \mathcal{SAA}\llbracket R \rrbracket \end{cases} \Bigg|\ \text{otherwise}$$

$$\mathcal{SAA}\llbracket (\ lb\ \texttt{<=}\ iv\ \texttt{<}\ ub)\ \mathit{block}\ R \rrbracket$$
$$= (\ lb\ \texttt{<=}\ \mathcal{SAA}'\llbracket \mathrm{iv} \rrbracket\llbracket \mathrm{ub} \rrbracket\ \texttt{<}\ ub)\ \mathcal{SAA}\llbracket \mathit{block} \rrbracket\ \mathcal{SAA}\llbracket R \rrbracket$$

$$\mathcal{SAA}'\llbracket iv \rrbracket\llbracket ub \rrbracket = \texttt{(1)}[\mathcal{S}(ub)]\ iv$$

$$\mathcal{SAA}'\llbracket [i_1,\ldots,i_d] \rrbracket\llbracket ub \rrbracket = [\texttt{(0)[[]]}\ i_1,\ldots,\ \texttt{(0)[[]]}\ i_d]$$

Fig. 4. Transformation scheme for inserting symbolic array attributes

denote the context-free replacement of a program fragment *expr* by a another program fragment *expr'*. The scheme $\mathcal{SAA}$ does not only express the relationships between the arguments and the shapes of the results of the SAC$_{\mathsf{mini}}$ built-in functions, but also ensures proper attribute annotation at function boundaries.

Identifiers bound to constants or applications of scalar-valued functions (including `dim`) are assigned the attribute pair `(0)[[]]`. Identifiers bound to the values of other identifiers (i.e. `a = b;`) share the same pair of attributes. By definition, the result of function `shape(`a`)` is a vector of length equal to the rank of a. This correspondence is expressed in the symbolic array attribute `(1)[[`$\mathcal{D}(a)$`]]`, which is converted into flat code to adhere to our grammar. Vice versa, the rank of the result of `reshape(`s`,`a`)` is determined by the length of vector s, which is accessed by selecting the first element from the shape of s.

The vector construct `[`a_0`,...,`a_{n-1}`]` yields an array whose rank is given by increasing the rank of a_0 by one. The shape vector is obtained by concatenating `[`n`]` and the shape vector of a_0[2] using the function `(++)` depicted in Fig. 3.

The WITH-loop `with ... genarray(`*shp*`,`*def*`)` generalises array construction. The rank of its result can be computed by adding the rank of the default value *def*[2] to the length of vector *shp*. Similar to vector construction, computing the shape vector requires to concatenate *shp* and the shape vector of *def*. The index vector is also annotated with symbolic array attributes by the auxiliary scheme $\mathcal{SAA}'$ before the main scheme is recursively applied to the parts. Here, we exploit the restriction that index vector and boundary vectors must coincide in length.

As explained so far, $\mathcal{SAA}$ inserts symbolic array attributes that describe an array's rank and shape in terms of existing arrays within the scope of the function body. For obvious reasons this approach can neither be carried over to function parameters nor to arrays defined by function applications. In both cases, we fall back to introducing applications of the built-in functions `dim` and `shape`. This is formalised by the auxiliary scheme $\mathcal{MIR}$ shown in Fig. 5. Scheme $\mathcal{R}$ only serves to provide fresh identifiers and thus avoid naming conflicts. The relationship between rank, shape and value of a function parameter or an application result is established by the additional application of an internal pseudo function `saabind`:

```
(d)[s] v' = saabind(d,s,v);
```

The assignment associates the identifier v' with the rank d and the shape s. However, it makes no statement about the array attributes of v which may not be present at all. Thus, the above line is substantially different from

```
(d)[s] v' = v;
```

which states that v' and v are identical and thus have the same attributes.

[2] By definition all elements of a vector must have the same shape. Likewise, all elements of an array created using a `genarray`-WITH-loop must match the default element in shape. If the compiler does not manage to guarantee this property by static analysis, the code generator inserts a run time check into compiled code. Thus, we may safely adopt one representative here, which is either the first element of a non-empty vector or the default element of a WITH-loop.

$$\mathcal{MIR}\llbracket\, t\,\texttt{[}s_1\texttt{,}\ldots\texttt{,}s_d\texttt{]}\;\; a \,\rrbracket = \begin{cases} \texttt{(0)[[]]} & a_d \;\texttt{=}\; d\texttt{;} \\ \texttt{(1)[[}d\texttt{]]} & a_s \;\texttt{=}\; \texttt{[}s_1\texttt{,}\ldots\texttt{,}s_d\texttt{];} \\ \texttt{(}d\texttt{)[[}s_1\texttt{,}\ldots\texttt{,}s_d\texttt{]]}\; a & \texttt{=}\; \texttt{saabind(}a_d\texttt{,}a_s\texttt{,}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \end{cases}$$

$$\mathcal{MIR}\llbracket\, t\,\texttt{[}\bullet_1,\ldots,\bullet_d\texttt{]}\;\; a \,\rrbracket = \begin{cases} \texttt{(0)[[]]} & a_d \;\texttt{=}\; d\texttt{;} \\ \texttt{(1)[[}d\texttt{]]} & a_s \;\texttt{=}\; \texttt{shape(}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \\ \texttt{(0)[[]]} & a_{s_1} \;\texttt{=}\; a_s\texttt{[0];} \\ \ldots & \\ \texttt{(0)[[]]} & a_{s_d} \;\texttt{=}\; a_s\texttt{[}d\texttt{-1];} \\ \texttt{(1)[[}d\texttt{]]} & a_s\texttt{'} \;\texttt{=}\; \texttt{[}a_{s_1}\texttt{,}\ldots\texttt{,}a_{s_d}\texttt{];} \\ \texttt{(}d\texttt{)[}a_s\texttt{]} & a \;\texttt{=}\; \texttt{saabind(}a_d\texttt{,}a_s\texttt{'}\texttt{,}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \end{cases}$$

$$\mathcal{MIR}\llbracket\, t\,\texttt{[*]}\;\; a \,\rrbracket = \begin{cases} \texttt{(0)[[]]} & a_d \;\texttt{=}\; \texttt{dim(}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \\ \texttt{(1)[[1]]} & a_{s_s} \;\texttt{=}\; \texttt{[}a_d\texttt{];} \\ \texttt{(1)[}a_{s_s}\texttt{]} & a_s \;\texttt{=}\; \texttt{shape(}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \\ \texttt{(}a_d\texttt{)[}a_s\texttt{]} & a \;\texttt{=}\; \texttt{saabind(}a_d\texttt{,}a_s\texttt{,}\mathcal{R}\llbracket\, a \,\rrbracket\texttt{);} \end{cases}$$

Fig. 5. Compilation scheme for representing array attributes at function boundaries

Depending on the shape class of the argument, different code patterns are used. For AKS arrays, both attributes are simple constants. This also holds for the rank attribute of AKD arrays. Their shape, however, must be determined dynamically. The elements of the shape vector are selected one by one and reassembled to form a new vector. Doing so we introduce mirrors for the whole shape vector and for all the elements which can be obtained by selecting from the new shape vector. In the AUD case, the pattern essentially applies `dim` and `shape`. The additional code line serves to encode the correspondence between the length of the shape vector and the rank of the array. The auxiliary scheme $\mathcal{T}$ used in the rule function application yields the base type of the function value.

4 Effects of Symbolic Array Attributes

Symbolic array attributes provide uniform access to the ranks and shapes of arrays even if these properties are unknown until run time. In conjunction, they reflect all static relationships between rank scalars, shape vectors and array values within one function. A compiler may now exploit this information as a foundation for program optimisation. In particular, symbolic array attributes allow the compiler to statically eliminate data dependencies resulting from accessing array rank and shape properties.

Fig. 6 shows the basic partial evaluation steps that exploit symbolic array attributes. All applications of `dim(`a`)` and `shape(`a`)` are replaced by the corresponding attribute values $\mathcal{D}(a)$ and $\mathcal{S}(a)$, respectively. Even applications of `saabind(`d`,`s`,`a`)` can actually be eliminated if there are symbolic array attributes for a identical to `(`d`)[`s`]`, which may well happen as a result of function inlining. Although these transformations seem simple, they are in fact crucial

$$\begin{array}{ll}
\mathcal{SVO}\,[\![\, \texttt{dim(}a\texttt{)} \,]\!] = \mathcal{D}(a) & \text{if } \mathcal{D}(a) \text{ defined} \\
\mathcal{SVO}\,[\![\, \texttt{shape(}a\texttt{)} \,]\!] = \mathcal{S}(a) & \text{if } \mathcal{S}(a) \text{ defined} \\
\mathcal{SVO}\,[\![\, \texttt{saabind(}a_d\texttt{,}a_s\texttt{,}a\texttt{)} \,]\!] = a & \text{if } \mathcal{D}(a) \equiv a_d \wedge \mathcal{S}(a) \equiv a_s
\end{array}$$

Fig. 6. Optimisation schemes for `dim`, `shape`, and `saabind`

for triggering a plethora of further optimisations that may even not be aware of symbolic array attributes.

By means of symbolic array attributes the available information on partial structural information of individual arrays as well as the structural relationships between different arrays are explicitly modeled in terms of regular SaC$_{\mathsf{mini}}$ expressions. As such they are subject to standard optimisations like constant folding, constant/variable propagation or common subexpression elimination to name just a few. Relationships between arrays like shape equality are expressed in the most natural way: by having all queries for the shape of one or another array be replaced by the same (symbolic) identifier. High-level optimisations like WITH-loop-folding [3] or WITH-loop-fusion [5] and memory management techniques like update-in-place or memory reuse [6] benefit from this information with little or no alteration. Likewise, *shape cliques* [9] can be identified without further analysis: all arrays belonging to the same shape clique have the same symbolic shape identifier.

Symbolic array attributes make exactly those rank and shape computations explicit in intermediate code that otherwise would be created by the code generator at a much later stage of the compilation process. If not a single optimisation applies, we end up with the same code generated in the end as without symbolic array attributes. However, our experience shows that typically our optimisations are quite effective, and, hence, rank and shape computation are partially performed at compile time and shared among different arrays in many cases.

5 Case Study

In this section, we demonstrate how symbolic array attributes influence the compilation process. Instead of quantifying performance using a broad range of benchmarks, our case study aims at illustrating in detail how program optimisation is affected and why symbolic array attributes allow us to generate more efficient code. For this purpose, we choose a small but very important example: element-wise mapping of a function to an array in a shape-generic way. In the absence of higher-order functions in SaC$_{\mathsf{mini}}$ (and in SaC), we need a concrete definition for each scalar operator. This functional pattern appears in abundance in generic SaC applications.

Fig. 7 shows the standard library implementation of extending the scalar boolean negation operator to boolean arrays of any shape. Essentially, the function `(!)` defines the result array to have the same shape as the argument array

```
bool[*] (!) ( int[*] a)
{
  s = shape(a);
  res = with (iv < s) {
          ael = a[iv];
          return( !ael);
        } genarray( s, false);
  return( res);
}
```

Fig. 7. Case study: element-wise mapping of a function to a generic array

```
bool[*] (!) ( int[*] a')
{
  (0) [[]]  ad  = dim(a');
  (1) [[1]] ass = [ad];
  (1) [ass] as  = shape(a');
  (ad)[as]  a   = saabind(ad,as,a');
  (1) [[1]] shs = [ad];
  (1) [shs] s   = shape(a);
  (0) [[]]  rd  = shs[0];
  (rd)[s]   res = with ((1)[shs] iv < s) {
                    (0)   [[]]  d   = shape(iv)[0];
                    (0)   [[]]  ss  = shape(s)[0];
                    (0)   [[]]  dl  = ss - d;
                    (1)   [[1]] ds  = [dl];
                    (0)   [[]]  dd  = shape(ds)[0];
                    (dd)  [ds] shp  = with ([(0)[[]]j] < ds) {
                                        (0)[[]] drel=s[j+d];
                                                return(drel);
                                      } genarray(ds,0);
                    (0)   [[]]  aed = shape(shp)[0];
                    (aed)[shp]  ael = with ((1)[dub] jv < shp) {
                                        (0)[[]] elem=sel(iv++jv,a);
                                                return(elem);
                                      } genarray(shp,0);
                              return( !ael);
                  } genarray( s, false);
            return( res);
}
```

Fig. 8. The example with inlined functions and symbolic shape attributes

with all elements being set to the negated values of the corresponding elements of the argument array. As the function signature deliberately leaves the structure of argument arrays unrestricted, selection into the argument array refers to the second instance of `select` from Fig. 2. Hence, it also relies on the auxiliary functions `drop` and `(++)` shown in Fig. 3.

Fig. 8 shows the intermediate code after inlining the functions `select` and `drop`. Due to the limited space we refrain from inlining the application of `++` as well. Without symbolic shape attributes no further optimisation would be possible. It is needless to say that this code shows a very poor run time performance. There is one WITH-loop alone for computing the shape `shp` of the selected element which is a relict from the `drop` function. Although it is bound to always yield the same result, the WITH-loop is evaluated for each element of the new array `res`. It cannot be lifted out of the outer WITH-loop because it depends on the index vector `iv` via `ds`, `dl` and `d`. Even worse, as the selected element is a scalar, `shp` must always be the empty vector `[]`. Hence, the following WITH-loop will only produce a single element by selecting into the array `a` at position `iv++[]` = `iv`.

However, annotating the code with symbolic array attributes, as described in Section 3, drives the optimisation process way beyond. The key to eliminating overhead in the outer WITH-loop lies in the highlighted code section in Fig. 8. By identifying that `iv` and `s` have in fact the same shape, it becomes apparent that the shape of the selected element is `[]`, i.e., the element turns out to be scalar. The symbolic shape attributes allow us to partially evaluate both `shape(iv)` and `shape(s)` to `shs`, such that both `d` and `s` become `shs[0]`, which is further resolved to `ad`. Exploiting the algebraic property that $x - x = 0$ makes `dl` become zero and thus `ds` turns into the constant vector `[0]`. Hence, standard optimisations transform the four highlighted lines of code in Fig. 8 into

```
(0) [[]]   d  = ad;
(0) [[]]   ss = ad;
(0) [[]]   dl = 0;
(1) [[1]]  ds = [0];
```

The optimisation process continues in a similar fashion. With `ds = [0]`, the inner WITH-loop that computes `shp`, the shape of the element selected from `a`, is known to merely yield the empty vector `[]`. As a consequence, the symbolic array attributes of `ael` have been refined to constants, namely `(0)[[]]`. Furthermore, with `shp = []`, the WITH-loop performing the selection itself can be unrolled, yielding `ael = sel(iv++[], a)`, which in turn is simplified to `ael = sel(iv, a)`.

Finally, by eliminating common subexpressions and dead code, we obtain the code shown in Fig. 9. The result looks strikingly similar to the original program in Fig. 7. However, instead of being forced to use the expensive generic selection function from Fig. 2, we now employ the built-in function `sel`. Moreover, the symbolic array attributes clearly reflect the shape equality between argument and result. This property is exploited by the compiler to generate code that tries to immediately reuse the memory that holds `a` for storing `res` [6].

In order to quantify the effect of the transformations enabled by symbolic array attributes in our case study, we have created a synthetic micro benchmark: We run 100 negations of an array of 2000 × 2000 elements[3] on a 3 GHz Intel Xeon processor. Fig. 10 shows program run times and memory consumption of the micro benchmark for compiled AKS, AKD and AUD code. Symbolic array

[3] SaC stores boolean values as integers rather than bits. Hence, we need approximately 16MB of memory to store one array.

```
bool[*] (!) ( int[*] a')
{
  (0) [[]]  ad  = dim(a');
  (1) [[1]] ass = [ad];
  (1) [ass] as  = shape(a');
  (ad)[as]  a   = saabind(ad,as,a');
  (ad)[as]  res = with ((1)[ass] iv < as) {
                    (0)[[]] ael = sel( iv, a);
                    return( !ael);
                  } genarray( as, false);
            return( res);
}
```

Fig. 9. The fully optimised example

attributes have no impact on the compilation of non-generic code. The AUD variant profits the most from the extended optimisation capabilities: execution time is reduced by 95% from 149.5s to 7.1s. This is not surprising given how much overhead has been eliminated from the intermediate program. The remaining slowdown with respect to the AKS program is explained by the lower efficiency of the AUD array traversal code. In both the AUD and the AKD case, symbolic array attributes enable memory reuse, thereby reducing space requirements to the AKS level. The AKD program especially benefits from this: its run time is reduced by 28%, approaching the AKS run time.

In generic array programming, small functions like the negation on arrays serve as building blocks for more complex operations. Fig. 11 illustrates this concept by means of a shape-generic implementation of element-wise logical implication. The function is composed of negation and disjunction, where the implementation of the latter follows the familiar pattern. Since our type system cannot express the shape conformability restriction on the argument arrays, we use an application of `reshape` instead. The run time figures show that symbolic array attributes have a drastic effect beyond the improvements we observed in the compilation of the individual components. Since the applications of `shape`

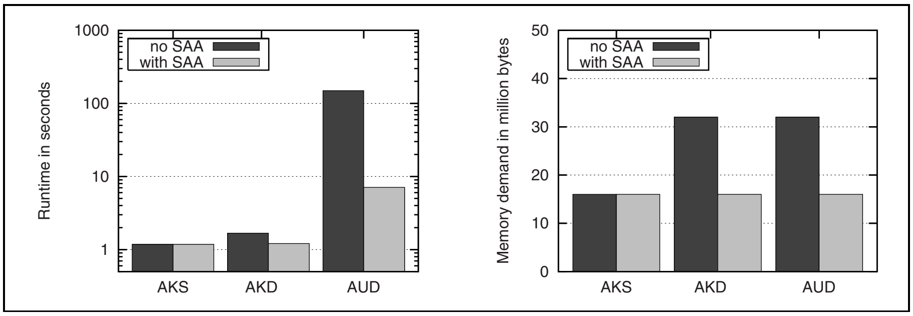

Fig. 10. Run time and memory impact of symbolic array attributes

```
bool[*] impl( bool[*] a, bool[*] b)
{
  a = reshape( shape(b), a);
  return( !a | b);
}
```

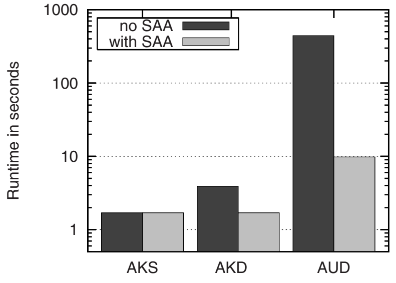

Fig. 11. Run time performance of logical implication on arrays

used in the constituent functions have been removed, WITH-loop-folding is able to merge the two consecutive WITH-loops performing negation and disjunction. Hence, the AKD execution time is reduced by 56% from 3.9s to 1.7s, once more reaching the performance level of the AKS variant. The AUD run time drops by almost 98% from 442.5s to 9.8s, making the once prohibitively expensive shape-generic program useful in practice.

6 Related Work

An example for the importance of structural information for array processing is Jay's FISh [10]. In FISh, each function `f` is accompanied by a shape function `#f` that maps the shape of the argument to the shape of the result. Shape inference proceeds by complete static evaluation of these shape functions and rejects all programs for which it fails. As a consequence, FISh does not support non-uniform functions like `take` and `drop` for which the result shape depends on argument values rather than only shapes.

SaC is less restrictive than FISh and properly supports non-uniform operations. However, efficiency of compiled code nevertheless depends on the accuracy of the available shape information [11]. To improve structural information we previously focused on a combination of partial evaluation and selective function specialisation [7]. Bernecky recently introduced the concept of *shape cliques*, sets of arrays of provably equal shape [9], and investigated their impact on a selected optimisation: index vector elimination. Symbolic array attributes generalise the concept of shape cliques by representing partial and relational structural information explicitly in the code. In particular, symbolic array attributes allow us to identify shape cliques, but go beyond this specific application.

Runtime performance is not a key issue in untyped, interpreted array languages like MatLab, APL or J. However, as soon as attempts are indeed made to accelerate program execution, structural array properties gain interest. For example, the Falcon MatLab compiler [12] by de Rose and Padua infers either precise shapes or rather fuzzy approximations like `notMatrix` and `notScalar`. Recently, Joisha and Banerjee [13] presented an approach for inferring symbolic array shapes that is based on modeling the shape semantics of the built-ins in an algebraic system and evaluation of the resulting expressions using term rewriting. Another approach taken by McGosh [14] is to use propositional logic to represent the constraints on the variables. The shape constraints of each statement are expressed as sequences of clauses, before a whole-procedure solution for all shapes is computed by finding *n-cliques* in the constraint graph. In the domain of APL Bernecky proposed array predicates [15] as a framework to represent knowledge about arrays that exceeds structural information, e.g., a vector may be attributed as *sorted* if it is the result of a sort operation. In a setting dominated by powerful built-in operations such predicates can be maintained and exploited at a later stage, e.g. to avoid (re-)sorting of an already sorted array.

All the approaches mentioned so far share with ours the aim to identify information that is hidden in the code. An alternative class of approaches enable the user to express constraints on arrays by more expressive type systems. Dependent types [8] naturally lend themselves for this purpose as they allow the use of (dynamic) terms to index within families of types. Unfortunately, the problem of type equality is generally undecidable as it boils down to deciding whether two index terms denote the same value. For example, Augustsson's Cayenne [16] is a fully dependently typed language. Its type system is undecidable and it lacks phase distinction. Both problems can be overcome by restricting the type language. For example, epigram [17,18] (Altenkirch, McBride, McKinna) rules out general recursion in type-forming expressions to retain decidability. Other, light-weight approaches such as Xi and Pfenning's dml [19], Xi's *applied type system* [20], and Zenger's *indexed types* [21] allow term-indexing into type families only for certain *index sorts*. The type-checking problem can then be reduced to constraint solving on these sorts, which is decidable.

Our work is in part inspired by the above mentioned dependently typed programming systems. Symbolic array attributes may be regarded as index-terms into the type family of multi-dimensional arrays of a given base type with SaC itself being used as the term language. However, our approach does not aim at providing stronger typing facilities, but at obtaining a uniform representation of the knowledge already present in a program. In consequence, there is no obligation of keeping type equality decidable. There is also a relation to work which aims at optimising dependently typed programs. Xi and Pfenning report successful array bounds check elimination [22], Xi even outlined a scheme for dead code removal through dependent types in dml [23]. McKinna and Brady describe optimisations in the compilation of epigram to remove compile time only values from terms as well as array bounds checks [24].

7 Conclusion and Future Work

We have proposed a novel approach to represent incomplete structural information on shape-generic arrays inferred by the compiler. The appealing characteristic of our symbolic array attributes is that they map information from the domain of shapely types into the domain of the expression language where a plethora of optimisation techniques wait to be reused to improve the compile time knowledge on structural properties of shape-generic arrays. As a consequence, we observe a cross-fertilisation between code optimisation and gathering of additional structural information. Our case study demonstrates how our technique may substantially improve the run time behaviour of shape-generic code without the need for specialisation into non-generic code. Although we have illustrated the concept of symbolic array attributes in the context of SaC, the ideas can be carried over to other settings with support for generic array programming rather straightforwardly.

A limitation of our approach so far is the fact that our analysis is mostly intra-functional. Function inlining and function specialisation with respect to symbolic array attributes are two ways to infer structural properties across function boundaries. However, both are somewhat orthogonal to our approach. Instead, we intend to embed our current work in a more general framework that actually extends the shape-generic type system by a variant of dependent types adapted to the special needs of shape-generic programming. This step would allow us to express structural relationships between function parameters and function results in a systematic way. Symbolic array attributes would then serve as an implementation vehicle for type inference and as an interface between the type system and the optimisation framework.

References

1. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming 13(6), 1005–1059 (2003)
2. Grelck, C., Scholz, S.B.: SAC — A Functional Array Language for Efficient Multi-threaded Execution. International Journal of Parallel Programming 34(4), 383–427 (2006)
3. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 72–92. Springer, Heidelberg (1998)
4. Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 118–134. Springer, Heidelberg (2004)
5. Grelck, C., Hinckfuß, K., Scholz, S.B.: With-Loop Fusion for Data Locality and Parallelism. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 178–195. Springer, Heidelberg (2006)
6. Grelck, C., Trojahner, K.: Implicit Memory Management for SAC. In: Grelck, C., Huch, F. (eds.) Hardware Specification, Verification and Synthesis: Mathematical Aspects. LNCS, vol. 408, Springer, Heidelberg (1990)
7. Grelck, C., Scholz, S.B., Shafarenko, A.: A Binding-Scope Analysis for Generic Programs on Arrays. In: Butterfield, A. (ed.) IFL 2005. LNCS, vol. 4015, pp. 212–230. Springer, Heidelberg (2006)
8. Martin-Löf, P.: Intuitionistic Type Theory. Biblio-Napoli (1984)

9. Bernecky, R.: Shape Cliques. In: Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation of Functional Languages, IFL 2006, Budapest, Hungary, September 4-6, 2006, Eötvös Loránd University 1–12 (2006)
10. Jay, C., Steckler, P.: The Functional Imperative: Shape! In: Hankin, C. (ed.) ESOP 1998 and ETAPS 1998. LNCS, vol. 1381, pp. 139–153. Springer, Heidelberg (1998)
11. Kreye, D.: A Compilation Scheme for a Hierarchy of Array Types. In: Arts, T., Mohnen, M. (eds.) IFL 2001. LNCS, vol. 2312, pp. 24–26. Springer, Heidelberg (2002)
12. de Rose, L., Padua, D.: Techniques for the translation of matlab programs into fortran 90. ACM Transactions on Programming Languages and Systems 21(2), 286–323 (1999)
13. Joisha, P., Banerjee, P.: An algebraic array shape inference system for matlab. ACM Transactions on Programming Languages and Systems 28(5), 848–907 (2006)
14. McCosh, C.: Type-based specialization in a telescoping compiler for matlab. Master Thesis TR03-412, Rice University, Houston, Texas, USA (2003)
15. Bernecky, R.: Reducing Computational Complexity with Array Predicates. In: Picchi, S., Micocci, M. (eds.) Proceedings of the International Conference on Array Processing Languages (APL'98), Rome, Italy, pp. 46–54. ACM Press, New York (1998)
16. Augustsson, L.: Cayenne – a language with dependent types. In: International Conference on Functional Programming. pp. 239–250 (1998)
17. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
18. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Manuscript, available online (2005)
19. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: Aiken, A. (ed.) Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), pp. 214–227. San Antonio, Texas, USA, ACM Press, New York (1999)
20. Xi, H.: Applied Type System (extended abstract). In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 394–408. Springer, Heidelberg (2004)
21. Zenger, C.: Indexed types. Theorectical Computer Science 187(1-2), 147–165 (1997)
22. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, pp. 249–257 (1998)
23. Xi, H.: Dead code elimination through dependent types. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 228–242. Springer, Heidelberg (1999)
24. McKinna, J., Brady, E.: Phase distinctions in the compilation of epigram. Draft, available online (2005)