Size Slicing - A Hybrid Approach to Size Inference in Futhark

Troels Henriksen Martin Elsman Cosmin E. Oancea
HIPERFIT, Department of Computer Science, University of Copenhagen (DIKU)
athas@sigkill.dk, mael@diku.dk, cosmin.oancea@diku.dk

Abstract

We present a shape inference analysis for a purely-functional language, named Futhark, that supports nested parallelism via array combinators such as `map`, `reduce`, `filter`, and `scan`. Our approach is to infer code for computing precise shape information at run-time, which in the most common cases can be effectively optimized by standard compiler optimizations. Instead of restricting the language or sacrificing ease of use, the language allows the occasional shape-dynamic, and even shape-misbehaving, constructs. Inherently shape-dynamic code is treated with a fall-back technique that preserves, asymptotically, the number of operations of the program and that computes and returns the array's shape alongside with its value. This approach leads to a shape-dependent system with existentially-quantified types, where static shape inference corresponds to eliminating existential quantifications from the types of program expressions.

We optimize the common case to negligible overhead via *size slicing*: a technique that separates the computation of the array's shape from its values. This allows the shape to be calculated in advance and to be used to instantiate the previously existentially-quantified shapes of the value slice. We report negligible overhead, on several mini-benchmarks and three real-world applications.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

General Terms Performance, Design, Algorithms

Keywords size analysis, functional language, dependent types

1. Introduction

Work on automatic parallelization in both functional [6] and imperative languages [7, 26] goes back at least to the late eighties and early nineties, but, while significant progress has been demonstrated at that time, these techniques have not achieved widespread use because parallel hardware was not yet mainstream.

The emergence of commodity multi-core, cache-coherent systems in mid 2000 has fostered the study (i) of software-transactional memories [13] (STM) as a way to provide a clean, progress-guaranteed semantics for atomic operations, (ii) of a variety of algorithms and transformations [32, 37] that were aimed at enhancing the locality of reference in both space and time, and (iii) of a range of analyses from entirely dynamic [11, 29, 34] to entirely static for automatic parallelization [20, 30, 33]. While these techniques are important and ideas can be reused, such solutions do not naturally extend to commodity (massively parallel) many-core architectures, such as, GPGPUs, because they (i) either rely on a fast and coherent cache infrastructure, (ii) exhibit memory overhead proportional to the number of cores, or (iii) do not extend beyond one-loop parallelization and do not guarantee that all available parallelism is detected. Furthermore, applications based on data-parallel programming APIs such as OpenCL [36] often obfuscate the original algorithm, inhibit compiler optimizations, and still reflect a time-stamped hardware.

Many-core (massively parallel) hardware is now mainstream and it raises both significant challenges, but also great opportunities for array languages to emerge as the prime technology for hardware-independent commodity programming. Futhark [21–23] is a (core) array language and compiler infrastructure that offers a unique blend of purely functional and imperative features: On the one hand, it supports nested parallelism on regular arrays (of tuples) via a set of second-order array combinators (SOAC) that have inherently parallel semantics (e.g., `map`, `reduce`, `filter`, `scan`). The rich algebra of invariants allows the compiler to implement aggressive code transformations, such as fusion. Further, Futhark supports (`do`) loops and in-place updates that still retain the pure-functional semantics: A loop is just a special case of a (tail) recursive function, and in-place updates are implemented via a uniqueness-type mechanism [1]. This combination allows us to express effectively (at least several) real-world applications for which (i) parallelism is made explicit at all levels (in the nest), and (ii) loops that exhibit cross-iteration array dependencies are represented efficiently.

Compiling a fairly high-level language such as Futhark towards low-level GPGPU code is not without challenges. For example, static (rather than dynamic) memory allocation is arguably better suited for the GPGPU architecture. However, Futhark, being designed for ease of use, does not require a programmer to make explicit the array shape at array-creation points, which sometimes is also not possible, as in the case of the use of `filter`.

This paper presents an analysis that infers the array shapes in Futhark programs. The analysis infers code for computing precise shape information as runtime values. The analysis targets the Futhark intermediate representation (IR), which means that it benefits from the standard Futhark compiler optimizations, such as inlining, common-subexpression elimination, fusion, and dead-code elimination. We model shape information via existentially-quantified (shape-)dependent types, which are represented in the target IR as integral values that are computed at the same time as the result arrays, that is, before the use of the arrays. This transformation preserves asymptotically the number of operations of the original program.

The analysis described so far would not be effective in the context of GPGPU execution, because, for example, the shape of

Submitted to FHPC'14, September 4, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3040-4/14/09…$15.00.
http://dx.doi.org/10.1145/2636228.2636238

an array produced by `map` would be known only after the execution of `map`, that is, the array would not be amenable to static allocation.

We optimize the common case via a *size-slicing* analysis, which separates the computation of the array's shape, from the computation of the array's values whenever a cost model allows it. The analysis eliminates the existential quantifiers of the (shape-)dependent type, because now the array's shape is available before the computation of the array elements. The current cost model allows size slicing whenever the shape computation requires $O(1)$ operations, that is, when no recurrences such as `map`, `loop`, or recursive functions appear in the shape slice.

Related dependent-type systems [38, 40, 41] may offer the programmer guarantees about the particular execution strategies implemented by a backend compiler, but this typically comes at the price of restricting the language and sacrificing ease of use. In comparison, we approach a full language, and accept the occasional shape-misbehaved construct. This is treated by the fall-back technique that uses existentially-quantified types, while we rely on size-slicing analysis and the common compiler infrastructure to optimise the common case.

Since Futhark is currently (only) interpreted, we performed a qualitative evaluation of (i) several micro-benchmarks, including matrix multiplication and Floyd's shortest path algorithm, and (ii) of three real-world applications [28] in the range of hundred to thousands of lines of code. Human inspection of the generated code shows that for all tested benchmarks the shape and bounds checking overhead is negligible; that is, it is asymptotically smaller than the symbolically computed number of operations (work) of the original program. For example, two real-world benchmarks exhibited an overhead on the order of hundreds of operations when the program's work was on the order of tens-of-millions of operations. All other benchmarks exhibited an overhead of $O(1)$ number-of-operations.

2. Compiler, Source and Target IR and Intuition

This section sets the stage for the presentation of the code transformation rules that implement size analysis: Section 2.1 briefly reviews the organization of the Futhark compiler. Section 2.2 presents the intermediate representation (IR), that is, the source language to size analysis, its types and typing rules. Section 2.3 presents the shape-dependent typing of the IR constructs, and how shapes and "assumed but not yet verified" invariants are made explicit in the target IR. Finally, Section 2.4 demonstrates the intuition behind our technique on a code example.

2.1 Futhark's Compiler Infrastructure

Futhark is a mostly-monomorphic, statically typed, strictly evaluated, purely functional language, intended effectively to support parallel programs to run on massively parallel hardware, such as GPGPUs. The language, named after the first six letters of the runic alphabet (i.e., "Fuþark") enables a regular notion of nested parallelism via a set of six second-order array combinators (SOACs): `map`, `reduce`, `filter`, `scan`, `multireduce` (not implemented yet), and `redomap` (the glue that enables (de)composability).

The compiler architecture is depicted in Figure 1. Type checking is performed on the original program to enable meaningful error messages, but normalisation renames all bindings to unique names and brings the program to an IR that resembles A-normal form [35], that is, three-address code of statements-like bindings. Additionally, tuples are flattened, in the sense that arrays-of-tuples are transformed to tuples-of-arrays [6] and tuples are expanded such that no variable is bound to a tuple value.

The simplification engine encompasses well established optimisations such as inlining, copy propagation, constant folding, common-subexpression elimination, dead-code elimination,

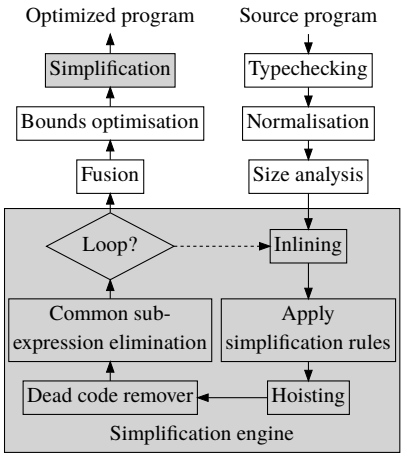

Figure 1. Compiler pipeline.

and hoisting of invariant terms out of recurrences in loops and SOACs. The simplification rules are critical for optimising the result of higher-level optimisations, such as producer-consumer fusion [21, 22], bounds-checking analysis [23], and size analysis, which is the subject of this paper. For example, bounds-checking and size analysis are implemented via high-level transformation rules that only guarantee that the asymptotic complexity (in number of operations) of the original program is preserved, but rely on the simplification engine to reduce the overhead to be negligible in the common case.

2.2 Source, Size-Agnostic Intermediate Representation

Whenever z is an object of some kind, we write $\bar{z}$ to range over sequences of objects of this kind. When we want to be explicit about the size of a sequence $\bar{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\bar{z}^{(n)}$ and we write $z, \bar{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$. We will write $\mathrm{d}(\tau)$ to indicate the *rank* (number of dimensions) of a type τ. We may also write $\mathrm{d}(v)$, where v is not itself a type, but something which *has* a type, such as a variable.

Figure 2 shows the (simplified) intermediate representation of Futhark which is the source language for size analysis:

- Variable names are ranged over by x, y, and s, and we use g and h to range over function names.
- A variable may have a scalar type (i.e., `bool`, `int`, `real`), or a multidimensional (regular) array type, such as [[`real`]], which is the type of a matrix, in which all rows have the same size.
- Named and unnamed functions use a syntax similar to SML, but unnamed functions may appear only as arguments to second-order array combinators, such as `mapT`, `reduceT`, `filterT`, and so on. Moreover, named functions may only be defined at top-level. Named functions may be mutually recursive.

Expressions in the intermediate language are normalized. That is, they consist of a sequence of `let`-bindings that always ends in a tuple of variable names. A pattern (p) consists of a (tuple of) typed variable name(s) that can be bound to:

- a constant, an array element (indexing), or a tuple of variables,
- an `if-then-else` expression or a function call,
- a loop that, using the notations from Figure 2, initialises a set of loop-variant variables $\bar{p}$ with the values of variables $\bar{x}$, and executes S iterations by re-binding $\bar{p}$ to the result of e_1. The

s, x ::= `id` (variable names, s for scalars/neutral elem)
g, h ::= `id` (function names)

t ::= `int | bool | real` (basic types)
τ ::= t (variables' types)
| $[\tau]$ (regular-array type)
ρ ::= $\{\tau_1, \ldots, \tau_n\}$ (tuple types)
ϕ ::= $\rho_1 \rightarrow \rho_2$ (fun/lambda type)

p ::= $\tau_1\ x_1, \ldots, \tau_n\ x_n$ (n-ary pattern)

e ::= $\{x_1, \ldots, x_n\}$ (tuple exp)
| `let` p `=` ct `in` e (constant binding)
| `let` p `=` $x[s_1, \ldots, s_n]$ `in` e (array indexing)
| `let` p `=` $\{x_1, \ldots, x_n\}$ `in` e (variable renaming)
| `let` p `=` $op(\bar{a})$ `in` e (operator call)
| `let` p `=` $g(\bar{x})$ `in` e (function-call)
| `let` p `=` `if` s `then` e_1 `else` e_2 `in` e_3 (if binding)
| `let` q `=` `loop` $(\bar{p}$ `=` $\bar{x})$ `for` $s < S$ `do` e_1 `in` e_2 (do-loop binding; $\bar{p}$ appears in e_1; $\bar{x}$ initialises $\bar{p}$)

a ::= x (simple argument)
| `fn` ρ $(p) \Rightarrow e$ (function argument)

P ::= `fun` ρ $g(p)$ `=` e; P (named function definition)
| `e` (main expression)

Figure 2. Source, Size-agnostic IR for Futhark.

loop has the semantics of the tail-recursive call:
`let` p `=` $h(0, S, x_1, \ldots, x_n)$ `in` e_2, where h is defined as:

```
fun {τ1,..., τn} h({int s, int S, τ1 p1,..., τn pn}) =
  if s >= S then {p1,...,pn} else h(s+1, S, e1)
```

- queries that return the `size` of a specific array dimension,
- operator calls, including calls to unary (e.g., `-, not`) or binary (e.g., `+, *`) operators, but also calls to a number of (polymorphic) array constructors and second-order combinators (SOAC).

For giving concise types to operators, we use a notion of *extended types* that supports polymorphism in types and for which arguments to functions may themselves be functions:

$\underline{\tau}$ ::= $t \mid \alpha \mid [\underline{\tau}] \mid \{\underline{\tau}_1, \ldots, \underline{\tau}_n\} \mid \underline{\tau}_1 \rightarrow \underline{\tau}_2$
$\underline{\sigma}$ ::= $\forall \bar{\alpha}.\underline{\tau}$

Extended types ($\underline{\tau}$) and extended type schemes ($\underline{\sigma}$) are used only for the treatment of operators and we shall be implicit about converting types and type schemes to and from their extended counter parts. A *substitution* (S) is a mapping from type variables to extended types. Applying a substitution S to some object B, written $S(B)$, has the effect of simultaneously applying S to type variables in B (being the identity outside its domain). An extended type $\underline{\tau}'$ is *an instance of* an extended type scheme $\underline{\sigma} = \forall \vec{\alpha}.\underline{\tau}$, written $\underline{\sigma} \geq \underline{\tau}'$, if there exists a substitution S such that $S(\underline{\tau}) = \underline{\tau}'$.

Type schemes for operators, including a representative subset of the SOAC operators, are given in Figure 4, and an informal description of them is given in Figure 3. The SOACs of the intermediate language (e.g., `mapT`) are the tuple-of-array version of the user language SOACs (e.g., `map`). The SOACs of the intermediate language receive an arbitrary number of array arguments and produce a tuple of arrays. As such, the semantics of a SOAC$\in$ {`mapT, scanT, filterT`} operator can be described as a composition between `unzip`, the user-language SOAC (e.g., `map`), and `zip`, where the unnamed function is suitably modified to work with the flat sequence of array arguments.

$op(a^{(n)})$	Description
$\ominus s$	Unary scalar operation on s.
$s_1 \oplus s_2$	Binary scalar operation on s_1 and s_2.
size(s,x)	Returns the size of dim s of x.
iota(x)	Returns the vector $[0, \ldots, x-1]$.
replicate(s,x)	Returns an array of rank one higher than x's rank, containing an s-times replication of x.
split(s,x)	Returns a pair of arrays resulting from splitting array x at index s.
concat(x_1,x_2)	Returns the array resulting from catenating the two arrays x_1 and x_2 along the last dimension.
mapT($\lambda, \bar{x}$)	Equivalent to $(\texttt{unzip}\ o\ \texttt{map}\ \lambda'\ o\ \texttt{zip})\bar{x}$, where `map(f,[`$x_1$`,..,`$x_n$`])`$\equiv$`[f(`$x_1$`),..,f(`$x_n$`)]` $\lambda \equiv \texttt{fn}\ \beta\ (p) \Rightarrow e, p \equiv \tau_1\ y_0, \ldots, \tau_n\ y_{n-1}$, $\lambda' \equiv \texttt{fn}\ \beta\ (\{\tau_1, .., \tau_n\}\ y) \Rightarrow \texttt{let}\ p = y\ \texttt{in}\ e$
reduceT($\lambda, \bar{s}, \bar{x}$)	Equivalent to $(\texttt{reduce}(\lambda', \bar{s}) o \texttt{zip})\bar{x}$, where `reduce(`$\odot$`,`$s$`,[`$x_1$`,..,`$x_n$`])`$= s \odot x_1 \odot .. \odot x_n$ $\lambda \equiv \texttt{fn}\ \bar{\tau}\ (p^1, p^2) \Rightarrow e, p^i \equiv \tau_1\ x_1^i, .., \tau_n\ x_n^i$, $\lambda' \equiv \texttt{fn}\ \bar{\tau}\ (\bar{\tau}\ x^1, \bar{\tau}\ x^2) \Rightarrow$ `let` $p_1 = x^1$ `in let` $p_2 = x^2$ `in` e
scanT($\lambda, \bar{s}, \bar{x}$)	Equiv. to $(\texttt{unzip}\ o\ \texttt{scan}(\lambda', \bar{s})\ o\ \texttt{zip})\bar{x}$
filterT($\lambda, \bar{x}$)	Equivalent to $(\texttt{unzip}\ o\ \texttt{filter}\ \lambda'\ o\ \texttt{zip})\ \bar{x}$

Figure 3. Description of SOAC operators.

op		TySch(*op*)
`size`	::	$\forall\alpha.\{\texttt{int}, \alpha\} \rightarrow \texttt{int}$
`iota`	::	$\texttt{int} \rightarrow [\texttt{int}]$
`replicate`	::	$\forall\alpha.\{\texttt{int}, \alpha\} \rightarrow [\alpha]$
`split`	::	$\forall\alpha.\{\texttt{int}, [\alpha]\} \rightarrow \{[\alpha], [\alpha]\}$
`concat`	::	$\forall\alpha.\{[\alpha], [\alpha]\} \rightarrow [\alpha]$
`mapT`	::	$\forall\bar{\alpha}^{(n)}\bar{\beta}^{(m)}.\{\{\bar{\alpha}^{(n)}\} \rightarrow \{\bar{\beta}^{(m)}\}, [\alpha_1], \ldots, [\alpha_n]\} \rightarrow \{[\beta_1], \ldots, [\beta_m]\}$
`reduceT`	::	$\forall\bar{\alpha}^{(n)}.\{\phi, \bar{\alpha}^{(n)}, [\alpha_1], \ldots, [\alpha_n]\} \rightarrow \{\bar{\alpha}^{(n)}\}$ where $\phi = \{\bar{\alpha}^{(n)}, \bar{\alpha}^{(n)}\} \rightarrow \{\bar{\alpha}^{(n)}\}$
`filterT`	::	$\forall\bar{\alpha}^{(n)}.\{\phi, [\alpha_1], \ldots, [\alpha_n]\} \rightarrow \{[\alpha_1], \ldots, [\alpha_n]\}$ where $\phi = \{\bar{\alpha}^{(n)}\} \rightarrow \texttt{bool}$

Figure 4. Type schemes for operators, including various SOACs.

Type environments (Γ) are finite maps from program variables to types or function types. When Γ is a type environment and p is a pattern $\tau_1\ x_1, \cdots, \tau_n\ x_n$, we write Γ, p to denote the typing environment $\Gamma, x_1 : \tau_1, \cdots, x_n : \tau_n$.

Typing rules for the source language are given in Figure 5. The rules allow inferences among sentences of the forms (1) $\vdash_{\mathrm{P}} p : \rho$, (2) $\Gamma \vdash_{\mathrm{a}} a : \tau/\phi$, (3) $\Gamma \vdash e : \rho$, and (4) $\Gamma \vdash_{\mathrm{P}} P : \rho$. Sentences of the first form are read "the pattern p is matched by expressions of tuple type ρ." Sentences of the second form are read "under the assumptions Γ, the operator argument a has type τ or function type ϕ." Sentences of the third form are read "under the assumptions Γ, the expression e has tuple type ρ." Finally, sentences of the fourth form are read "under the assumptions Γ, the program P has tuple type ρ."

2.3 Shape-Dependent Typing and Invariants

One important observation is that some operator-semantics invariants, related to the array regularity, are guaranteed to hold by construction, but several other invariants are only "assumed", that is, they have not been verified (made explicit) in the IR:

- `iota`, `replicate`, and `split` assume a positive first argument.

Patterns $\boxed{\vdash_{\mathrm{p}} p : \rho}$

$$\frac{}{\vdash_{\mathrm{p}} \tau_1\ x_1, \cdots, \tau_n\ x_n : \{\tau_1, \cdots, \tau_n\}} \quad (1)$$

Operator arguments $\boxed{\vdash_{\mathrm{a}} a : \tau/\phi}$

$$\frac{}{\Gamma \vdash_{\mathrm{a}} x : \Gamma(x)} \quad (2)$$

$$\frac{\vdash_{\mathrm{p}} p : \rho \quad \Gamma, p \vdash e : \rho'}{\Gamma \vdash_{\mathrm{a}} \mathtt{fn}\ \rho'\ (p) \Rightarrow e : \rho \to \rho'} \quad (3)$$

Expressions $\boxed{\Gamma \vdash e : \rho}$

$$\frac{}{\Gamma \vdash \{x_1, \cdots, x_n\} : \{\Gamma(x_1), \cdots, \Gamma(x_n)\}} \quad (4)$$

$$\frac{\Gamma(y) = [\tau] \quad \Gamma(s) = \mathtt{int} \quad \Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \mathtt{let}\ \tau\ x = \ y[s]\ \mathtt{in}\ e : \rho} \quad (5)$$

$$\frac{\mathrm{ConstType}(ct) = \tau \quad \Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \mathtt{let}\ \tau\ x = \ ct\ \mathtt{in}\ e : \rho} \quad (6)$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{a}} a_i : \underline{\tau}_i \quad i = [0; n[\\ \mathrm{TySch}(op) \geq \{\bar{\underline{\tau}}^{(n)}\} \to \rho \quad \vdash_{\mathrm{p}} p : \rho \quad \Gamma, p \vdash e : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ p = \ op(\bar{a}^{(n)})\ \mathtt{in}\ e : \rho'} \quad (7)$$

$$\frac{\begin{array}{c}\Gamma(x_i) = \tau_i \quad i = [0; n[\\ \Gamma(g) = \{\bar{\tau}^{(n)}\} \to \rho \quad \vdash_{\mathrm{p}} p : \rho \quad \Gamma, p \vdash e : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ p = \ g(\bar{x}^{(n)})\ \mathtt{in}\ e : \rho'} \quad (8)$$

$$\frac{\begin{array}{c}\Gamma(s) = \mathtt{bool} \quad \Gamma \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \rho \\ \vdash_{\mathrm{p}} p : \rho \quad \Gamma, p \vdash e : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ p = \ \mathtt{if}\ s\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{in}\ e : \rho'} \quad (9)$$

Programs $\boxed{\Gamma \vdash_{\mathrm{P}} P : \rho}$

$$\frac{\begin{array}{c}\vdash_{\mathrm{p}} p : \rho_1 \quad \Gamma, p \vdash e : \rho_2 \\ \Gamma(g) = \rho_1 \to \rho_2 \quad \Gamma \vdash_{\mathrm{P}} P : \rho\end{array}}{\Gamma \vdash_{\mathrm{P}} \mathtt{fun}\ \rho_2\ g(p)\ = \ e; P : \rho} \quad (10)$$

$$\frac{\Gamma \vdash e : \rho}{\Gamma \vdash_{\mathrm{P}} e : \rho} \quad (11)$$

Figure 5. Typing rules for the source language.

- `concat` operator assumes that the arguments have identical shape for all but the outermost dimensions, and guarantees that the size of the outermost dimension of the result is the sum of the outermost dimension sizes of the arguments.

$\tau ::= \quad t \mid [\tau, s]$ (outer array of size s)

$\phi ::= \quad \forall \bar{s_1}.(\bar{\tau_1}) \to \exists \bar{s_2}.\bar{\tau_2}$ (sizes of args $\in \bar{s_1}$, results $\in \bar{s_2}$)

`concat` :: $\forall\ s_1\ s_2\ \tau.\ \{[\tau, s_1], [\tau, s_2]\}\ \to\ [\tau, s_1 + s_2]$

`reduceT` :: $\forall\ s\ \bar{\alpha}^{(n)}.\{\phi, \bar{\alpha}^{(n)}, [\alpha_1, s], \ldots, [\alpha_n, s]\} \to \{\bar{\alpha}^{(n)}\}$
where $\phi = \{\bar{\alpha}^{(n)}, \bar{\alpha}^{(n)}\} \to \{\bar{\alpha}^{(n)}\}$

`filterT` :: $\forall\ s_1\ \bar{\alpha}^{(n)}.\{\phi, [\alpha_1, s_1], \ldots, [\alpha_n, s_1]\}$
$\to\ \exists\ s_2.\ \{[\alpha_1, s_2], \ldots, [\alpha_n, s_2]\}$
where $\phi = \{\bar{\alpha}^{(n)}\} \to \mathtt{bool}$ and $0 \leq s_2 \leq s_1$

`mapT` :: $\forall \bar{t}^{(n+m)}\ s\ \bar{s_\alpha}.\exists \bar{s_\beta}.\ \{\ (\bar{\alpha}^{(n)} \to \bar{\beta}^{(m)}),$
$\{[\alpha_1, s], \ldots, [\alpha_n, s]\}\ \}$
$\to\ \{[\beta_1, s], \ldots, [\beta_m, s]\},$
where $\beta_i = [\ldots[t^b_{i+n}, s_1^{\beta_i}]\ldots, s_{k_i}^{\beta_i}]$, and $s_j^{\beta_i} \in \bar{s_\beta}$, $\forall i \in \{1 \ldots m\},\ j \in \{1 \ldots k_i\}$. Similar for α.

Figure 6. Dependent-size types for various SOACs.

- `mapT` is guaranteed to receive arguments of identical outermost size, which also matches the outermost size of all result arrays[1]. However, `mapT` assumes that its function argument produces arrays of identical shape for each element of the input array.
- `filterT` receives and produces arguments and results of identical outermost size, respectively (and the outermost size of the argument is greater than the one of the result).
- `reduceT` and `scanT` receive arguments of identical outermost size, and `scanT` results have outermost size equal to that of the input. The semantics for `reduceT` and `scanT` assumes that the (corresponding two) arguments and result of the (tuple-flattened) binary associative operator have identical shapes.

Figure 6 shows an extended type system in which (i) sizes are encoded in each array type, that is, $[\tau, s]$ represents the type of an array in which the outermost dimension has size s, and in which (ii) function/lambda types use universal quantifiers for the sizes of the array parameters ($\forall s_1$), and existential quantifiers for the sizes of the result arrays ($\exists s_2$). Figure 6 shows that this extension allows to encode the afore-mentioned invariants into size-dependent types.

The type of `mapT` is verbose because the result array types cannot be declared entirely in either the universally or existentially quantified parts. As such, the result type is expressed in terms of the (i) universally quantified basic-array types $\bar{t}^{(n+1:n+m)}$ and outermost size s, and (ii) the existentially quantified inner-dimension sizes $\bar{s_\beta}$. The next section demonstrates, by a code example, the code transformation that (i) makes explicit in the code the shape-dependent types and verifies the assumed invariants and (ii) optimizes away in many cases the existential types.

2.4 Birds-Eye View of the Approach

The program in Figure 7 receives as input a three-dimensional array `A`, and produces a two-dimensional array `B`, by mapping the elements of the outermost dimension of `A` by function `f`, such that each of the rows of `B` is the catenation of the arrays produced by reducing the innermost dimension of `A` by addition and multiplication, respectively. The Futhark user language is very close to the source IR in Figure 7, except that types are required only in function declarations, and expressions need not be in three-address form.

[1] In the user language `zip` accepts an arbitrary number of array arguments that are required to have the same outermost size.

```
fun [real] f([[real]] a) =
  let [real] S, [real] P =
    mapT( fn {real,real} ([real] x) =>
          let real s = reduceT(op +, 0.0, x) in
          let real p = reduceT(op *, 1.0, x) in
            {s,p}
        , a ) in
  let [real] R = concat(S, P) in
  R

fun [real] main([[real]]] A)=
  let [[real]] B = mapT(fn [real] ([[real]] a)=>f(a), A)
  in  B[5]
```

Figure 7. Running example: Program in source IR.

```
fun {int, [real,?0]} f(int N, int M, [[real,M],N] a) =
  let [real,N] S, [real,N] P =
    mapT( fn {real,real} ([real,M] x) =>
          let real s = reduceT(op +, 0.0, x) in
          let real p = reduceT(op *, 1.0, x) in
            {s,p}
        , a ) in
  let k = N + N in
  let [real,k] R = concat(S, P) in
  {k, R}

fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if K == 0 then 0
          else let s, _ = f(N, M, a[0]) in
               s
  let [[real,S],K] B =
    mapT(fn [real,S] ([[real,M],N] a) =>
           let int s, [real,s] v = f(N, M, a) in
           let c = assert(s == S) in
           <c>v,
         A)
  in  B[5]
```

Figure 8. Running example: ∃-quantified target IR.

The parameters/result of `main` are treated in adhoc fashion (read/written from/to a file). Note that the source IR is size-agnostic (typed): for example, `B` has been inferred to be a two-dimensional array of reals - as a result of a `map` with a function of signature `[[real]]` → `[real]` - but its shape is not known yet.

The first stage, demonstrated in Figure 8, transforms the program into an unoptimised version in which (i) all arrays have shape-dependent types, which may be existentially quantified, and (ii) all "assumed" invariants are explicitly checked. This is achieved by:

- extending the function signatures to encompass also the shape information for each array argument, e.g., `f` receives additional arguments `N` and `M` that specify the shape of array argument `a`,
- using whenever possible the "guaranteed" invariants to determine the shape of a result array, e.g., since array `a` (in function `f`) has outermost dimension of size `N`, then applying `mapT` to `a` will result in arrays `S` and `P` of (outermost) size `N`, and as such, `concat(S,P)` will produce an array of size `N+N`.
- representing function's array results via existentially-quantified shape-dependent types. The latter corresponds to modifying the function's body to also return the shape of the result array. For example, the result of `f` is now `{k, R}` where `k` is the size of `R`, and the result type of `f` is `{int, [real,?0]}` denoting that the size of the array result is the first `int` result, and finally,
- systematically inserting code that verifies all "assumed" invariants. For example, verifying the regularity of array `B` corresponds to checking that the size `s` of each array obtained from a call to `f` (inside `mapT`), is equal to `S`. The latter `S` is the predicted size of the innermost dimension of `B`, and has been computed before the execution of `mapT` by applying `mapT`'s lambda to `A[0]` (Otherwise, if `A` is empty then `S=0`; note also that the translation of `mapT` does not introduce existential types.) The syntax `<c>v` signifies that `v` may only be returned if the assertion corresponding to `c` succeeded.

```
fun int f_shape(int N, int M, [[real,M],N] a) =
  N + N

fun [real,res]
    f_value(int res, int N, int M, [[real,M],N] a) =
  let [real,N] S, [real,N] P =
    mapT( fn {real,real} ([real] x) =>
          let real s = reduceT(op +, 0.0, x) in
          let real p = reduceT(op *, 1.0, x) in
            {s,p}
        , a ) in
  let [real,res] R = concat(S, P) in
  R

fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if N == 0 then 0
          else f_shape(N,M,A[0]) in
  let [[real,S],N] B =
    mapT(fn [real,S] ([[real,K],M] a) =>
           let int s = f_shape(N,M,a) in
           let [real,s] v = f_value(s,N,M,a) in
           let c = assert(s == S) in
           <c>v,
         A)
  in  B[5]
```

Figure 9. Extracting and simplifying the shape slice of `f`

It is important to note that this transformation preserves asymptotically the number of operations of the original program.

The main technique used to eliminate existentially-quantified types from a function signature is to *split* such a function into a *value* and a *shape* version, which return (only) the original array results and their shapes, respectively. The splitting is decided by a cost model. The current one simply performs splitting only if the (simplified) shape version contains no recurrence constructs, e.g., `loop`, `mapT`, recursive function calls. (This is checked automatically by the compiler.) Figure 9 demonstrates the technique:

The shape and value slices of `f` are `f_shape` and `f_value` (after simplification). One can observe that the splitting succeeds because `f_shape` costs one addition, and that `f_value` receives one extra parameter, named `res`, that represents its result-array shape. Finally, in the body of `main`, the call to `f` in `mapT`'s lambda has been replaced with consecutive calls to `f_shape` and `f_value`.

Splitting the original computation into value and shape slices enables powerful restructuring transformations, for example, the companion paper [23] presents how the "assumed" invariants can be separated from the original computation under the form of a predicate, which guards the original computation, and which is aggressively optimised. This technique is demonstrated in Figure 10:

The original `mapT` from `main` is split into a predicate slice and a value slice. The predicate slice corresponds to the (first) `mapT` in Figure 10.*A* that produces a one-dimensional array of boolean values recording whether each of the original calls to `f` (in `mapT`'s lambda) produces an array of size `S`. If any of these values are `False`, then a runtime error will be generated (by `assert`), otherwise, if all are `True`, i.e., `reduce(&&,True,bs)` holds, then the program is valid and the execution of the computational slice can proceed. The latter corresponds to the second `mapT` in Figure 10.*A*,

A. Distributing out assertions...

```
fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if N == 0 then 0
          else f_shape(N,M,A[0]) in
  let [bool,N] bs =
    mapT(fn cert ([[real,M],N] a) =>
           let int s = f_shape(N,M,a) in
           s == S,
         A)
  let c = assert(reduce(&&, True, bs)) in
  let [[real,S],N] B =
    <c>mapT(fn [real,S] ([[real,M],N] a) =>
              f_value(S,N,M,a)
         A)
  in  B[5]
```

B. Inlining `f_shape`...

```
fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if N == 0 then 0
          else N+N in
  let [bool,N] bs =
    mapT(fn cert ([[real,M],N] a) =>
           let int s = N+N in
           s == S,
         A)
  let c = assert(reduce(&&, True, bs)) in
  ...
```

C. Rewriting `mapT` *to* `replicate`...

```
fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if N == 0 then 0
          else N+N in
  let [bool,N] bs = replicate(N, N+N == S)
  let c = assert(reduce(&&, True, bs)) in
  ...
```

D. Simplifying reduction...

```
fun [real,?0] main(int K, int N, int M, [[real,M],N],K] A)=
  let S = if N == 0 then 0
          else N+N in
  let c = if N == 0 then assert(True)
          else assert(N+N == S) in
  let [[real,S],N] B =
    <c>mapT(fn [real,S] ([[real,M],N] a) =>
              f_value(S,N,M,a)
         A)
  in  B[5]
```

Figure 10. Running example: Optimized (∃-free) target IR.

inside which only `f_value` is used because the shape of its result is known to be `S`, i.e., has been already computed and verified.

Figure 10.*B* shows the predicate slice after inlining `f_shape`. Since the body of the (first) `mapT` lambda is invariant to the lambda's arguments, the `mapT` is simplified to `replicate(N, N+N==S)` in Figure 10.*C*. Finally, in Figure 10.*D*, reducing the array obtained from `replicate` with the logical-and operator has been simplified to checking once `N+N == S`. The latter two steps are only possible because `f_shape` is particularly simple - specifically, it does not use the elements of the argument array. However, prior to exploiting this property, we were still able to check this assertion via a more expensive, map-reduce construct.

One may observe that in the resulted code, the shape and regularity of `B` are computed and verified before the definition of `B`, respectively, and, most importantly, that the size computation and assumed-invariant verification introduce negligible overhead, i.e., $O(1)$ number of operations. This is essential for GPGPU execution

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{<}\bar{s}^{(m)}\texttt{>}x_1, \ldots, x_n & \text{(predicated tuple exp)} \\
 & | & \texttt{let}\ p = \texttt{<}\bar{s}^{(n)}\texttt{>}op(\bar{a})\ \texttt{in}\ e & \text{(predicated call)} \\
 & | & \texttt{let}\ p = \texttt{assert}(a)\ \texttt{in}\ e & \text{(assertion)}
\end{array}
$$

Figure 11. Certificates and assertions.

because dynamic allocation and assertions are typically not well suited for accelerators, hence the shapes of the result and of various intermediate arrays need to be computed (or at least overestimated) and verified before the kernel is run.

However, it is not always the case that the technique that eliminates the existential quantifier (by slice separation) also preserves asymptotically the number of operations of the program. For example separating a recursive function might square up the original number of operations, as the shape function may itself end up recursive, and the value function may call it for every level of the recursion. In such cases we do not perform slicing and work with the existentially-quantified program. The next section presents in detail the transformation rules, and several corner cases.

3. Transformation rules

This section presents a set of syntax-directed rules for transforming an un-annotated Futhark program into an *annotated* Futhark program, where all types (except return types of top-level functions) have full shape annotations, and we explicitly check the regularity assumptions given for `mapT` in the previous section. Checks for other invariants – such as checking the inner sizes of the operands to `concat`) – are elided for brevity, but are present in the actual implementation.

In order to perform these checks, we formally introduce the `assert` construct mentioned in Section 2.4. An `assert`(s) expression, where s is of type `bool`, returns s, except that if the value of s is false, program execution halts with an error. An operator invocation or function return can be *predicated* on a sequence of variable names even if said variables are not otherwise used in the computation, which prevents the expression computing said variables from being removed as dead code. This is used to ensure that `assert` expressions stay in the program.

3.1 Fundamental Transformation

For each function f in the original program, we generate the *existential function* f_{ext}, that returns the values returned by f, with all shapes in the return type being existentially quantified.

Specifically, if the return type of a function f is $\{\tau_1, \cdots, \tau_n\}$, then the return type of f_{ext} is $\{\overline{\texttt{int}}^{\text{d}(\tau_1)}, \cdots, \overline{\texttt{int}}^{\text{d}(\tau_n)}, \tau_1', \ldots, \tau_n'\}$, where $\text{d}(\tau)$ is the rank of τ, and τ_i' is τ_i shape-annotated with $\overline{\texttt{int}}^{\text{d}(\tau_i)}$. For example, if f returns type $\{[[\texttt{int}]]\}$, f_{ext} will return type $\{\texttt{int}, \texttt{int}, [[\texttt{int}, ?1], ?0]\}$. Here, ?0 refers to the first `int` return, and ?1 to the second. Thus, after transformation, the shape of the return of a function will be existentially quantified.

Furthermore, the parameters of f are likewise annotated. An explicit `int` parameter is added for every dimension of an array parameter, with the array parameter itself annotated to refer to the corresponding `int` parameter. For example, if f takes a single parameter $[[\texttt{int}]]p$, then f_{ext} will take three parameters `int` n, `int` m, and $[[\texttt{int}, m], n]$.

This type translation corresponds exactly to making explicit the existentials of the dependent return type of f, which in the above example would be

$$\forall n.\forall m.[[\texttt{int}, m], n] \rightarrow \exists p.\exists l.[[\texttt{int}, l], p],$$

as actual values being passed around in the program.

$$\mathcal{T}(\tau, \bar{s}^{(\mathrm{d}(\tau))}) = \quad \texttt{[[[t,}s_{\mathrm{d}(\tau)}\texttt{],}\cdots\texttt{],}s_1\texttt{]}$$
$$\mathcal{T}(t, \bar{s}^{(\mathrm{d}(t))}) = \quad t$$

$$\mathcal{V}(\tau\ x, \bar{s}^{(\mathrm{d}(\tau))}) = \quad \mathcal{T}(\tau, \bar{s}^{(\mathrm{d}(\tau))})\ x$$

Where t is the basic type of τ and $\mathrm{d}(\tau)$ is the rank of τ.

Figure 12. Annotating bindings and types.

The following section will describe how to *slice* an existentially typed function into shape and value functions, and Sections 3.2-3.4 present some of the transformation rules that move from unannotated to *existentially*-annotated Futhark.

We will use function $\mathcal{V}(\tau\ x, s)$ in Figure 12 to annotate a binding $\tau\ x$ with the shapes in s, which is a sequence of variable names whose length is equal to the rank of τ. Similarly, $\mathcal{T}(\tau, s)$ performs this annotation on a type. As an example,

$$\mathcal{T}([[\texttt{int}]], \texttt{n}, \texttt{m}) = [[\texttt{int}, \texttt{m}], \texttt{n}]$$

and

$$\mathcal{V}([[\texttt{int}]]\ \texttt{a}, \texttt{n}, \texttt{m}) = [[\texttt{int}, \texttt{m}], \texttt{n}]\ \texttt{a}.$$

3.1.1 Size Slicing

We assume we have an existential function f_{ext} that has the form:

$$\begin{array}{l}\texttt{fun}\ \{\overline{\texttt{int}}^{\mathrm{d}(\tau_1)}, \cdots, \overline{\texttt{int}}^{\mathrm{d}(\tau_n)}, \tau'_1, \ldots, \tau'_n\}\ f_{\mathrm{ext}}(\overline{p}^m) = \\ \ \texttt{let}\ bnds\ \texttt{in} \\ \ \{\overline{s}^{\mathrm{d}(\tau_1)}, \cdots, \overline{s}^{\mathrm{d}(\tau_n)}, \overline{x}^n\}\end{array}$$

This function takes m parameters, and returns n interesting values (the xs), with the the ss being the shapes. One can trivially derive the corresponding shape function, by simply removing the values from the return type and result expression:

$$\begin{array}{l}\texttt{fun}\ \{\overline{\texttt{int}}^{\mathrm{d}(\tau_1)}, \cdots, \overline{\texttt{int}}^{\mathrm{d}(\tau_n)}\}\ f_{\mathrm{shape}}(\overline{p}^m) = \\ \ \texttt{let}\ bnds\ \texttt{in} \\ \ \{\overline{s}^{\mathrm{d}(\tau_1)}, \ldots, \overline{s}^{\mathrm{d}(\tau_n)}\}\end{array}$$

For the value function f_{value}, we introduce a number of new parameters, corresponding to the original existential shape, that give the result shape. The idea is that these parameters are the result of a call to f_{shape}, but this is not verifiable by the compiler. This may at first seem fragile, but it is worth keeping in mind that this transformation is done by the optimiser internally, and is not a user-visible feature. Hence, human error (apart from errors in the compiler implementation) will not be a factor. We use these new parameters to annotate the return type of the value function, i.e., the as' of f_{value} are instantiated with the results of f_{shape}:

$$\begin{array}{l}\texttt{fun}\ \{\mathcal{T}(\tau_1, \overline{a}^{\mathrm{d}(\tau_1)}), \ldots, \mathcal{T}(\tau_n, \overline{a}^{\mathrm{d}(\tau_n)})\} \\ \quad f_{\mathrm{value}}(\overline{a}^{\mathrm{d}(\tau_1)}, \ldots, \overline{a}^{\mathrm{d}(\tau_n)}, \overline{p}^m) = \\ \ \texttt{let}\ bnds\ \texttt{in} \\ \ \{\overline{x}^n\}\end{array}$$

Usually, there will be a large amount of dead and unnecessary code in f_{shape}, as most of the bindings will be concerned with computing values, even though the function is only returning shapes. This is for example the case with the shape function generated in Figure 9, where almost the entire body is removed by simplification. In practise, sophisticated dead code removal may be necessary to obtain efficient shape functions – for example, we will need to remove shape-invariant loops – and thus our approach generally requires the compiler to possess an effective simplification engine.

3.2 Transformation Functions

The function $\mathcal{A}^{\mathrm{exp}}_{\Sigma}(b)$ computes the *annotated* version of the body b in the environment Σ and returns shapes as well as values (that is, its type will contain existentials). The environment Σ is a mapping from names of arrays to lists of variable names, where element i of the list denotes the size of dimension i of the corresponding array. We will use conventional head, tail and drop operations to manipulate these lists, as well as bracket notation for arbitrary indexing; we write the "cons" operation as $x :: xs$.

The function $\mathcal{A}^{\mathrm{fun}}(f)$ computes the existentially-quantified function f_{ext}, by using $\mathcal{A}^{\mathrm{exp}}_{\Sigma}$ to annotate the function's body (and result) with shapes information, and by modifying the function's type as described in the beginning of Section 3.1.

We also have a function $\mathcal{A}^{\mathrm{lam}}_{\Sigma}(\lambda, \overline{r}, \overline{p})$. This is similar to the function transformation, except that (i) we work within an environment Σ, and (ii) we know in advance the intended shape of the result ($\overline{r}$) and parameters ($\overline{p}$), because the result shape of an anonymous function is never existentially quantified.

For clarity of presentation, we elide type annotations for bindings where the shape and type is obvious or not important; in the implementation, however, all bindings have full shape annotations.

3.3 Simple Rules

This section describes cases for the function $\mathcal{A}^{\mathrm{exp}}_{\Sigma}(b)$.

As the nonrecursive case, when given a leaf tuple expression, we simply look up the shapes of the returned values in the environment, and return those alongside the values:

$$\begin{array}{l}\mathcal{A}^{\mathrm{exp}}_{\Sigma}(\{\overline{x}^{(n)}\}) = \{\overline{x_1}, \cdots, \overline{x_n}, x_1, \cdots, , x_n\} \\ \ \text{where}\ \ \overline{x_1} = \Sigma(x_1) \\ \qquad\qquad \cdots \\ \qquad\qquad \overline{x_n} = \Sigma(x_n)\end{array}$$

Other simple cases are listed in Figure 13. Note that, for `concat` and `split`, we introduce new bindings in the program to record the outer sizes of the resulting arrays. Also, observe how `size` expressions are completely removed from the annotated program, and instead replaced with the variable storing the desired size.

A call to a function f becomes a call to the function f_{ext}, where argument sizes are passed explicitly, and the shapes of return values are existentially quantified. Concretely, we have a single binding group, where the existential context (the shapes) are bound first, followed by the actual values. In the resulting program, there is no syntactic distinction between the context and the values, and both can be used as ordinary variables from that point in the program. This rule is shown on Figure 14. In Section 4, we will describe an optimisation which will split this binding into two distinct bindings - one for the shape context, and one for the values - whenever an efficient f_{shape} exists for f_{ext}.

Annotating an `if-then-else`-expression is surprisingly tricky:

1. If the shape of the result can be known in advance, we wish for the type to not be existentially quantified, i.e., have no bindings of shape variables.
2. Yet, one of the two branches may be "more existential" than the other, in the sense that for one branch, we may in advance know the shape of the result, but on the other, we will need existential quantification. In this case, we must be conservative, and existentially quantify all dimensions of the result that are existential in either of the two branches.

However, for the initial transformation given on Figure 15, we will simply consider every shape in the result to be existentially quantified. In Section 4 we will cover how to optimise this further.

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v \texttt{ = iota(}n\texttt{) in } e) =$$
$$\quad \texttt{let [int,}n\texttt{] } v \texttt{ = iota(}n\texttt{) in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v \mapsto \langle n \rangle, \Sigma}(e)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v \texttt{ = replicate(}n\texttt{, } a\texttt{) in } e) =$$
$$\quad \texttt{let } \mathcal{V}(v, n :: \Sigma(a)) \texttt{ = iota(}a\texttt{) in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v \mapsto n :: \Sigma(a), \Sigma}(e)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v \texttt{ = concat(}a_1, a_2\texttt{) in } e) =$$
$$\quad \texttt{let } n \texttt{ = head}(\Sigma(a_1)) \texttt{ + head}(\Sigma(a_2)) \texttt{ in}$$
$$\quad \texttt{let } \mathcal{V}(v, n :: \text{tail}(\Sigma(a_1))) \texttt{ = concat(}a_1, a_2\texttt{) in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v \mapsto n :: \text{tail}(\Sigma(a_1)), \Sigma}(e)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v_1, v_2 \texttt{ = split(}n, a\texttt{) in } e) =$$
$$\quad \texttt{let } m \texttt{ = head}(\Sigma(a)) \texttt{ - } n \texttt{ in}$$
$$\quad \texttt{let } \mathcal{V}(v_1, n :: \text{tail}(\Sigma(a))), \mathcal{V}(v_2, (m, \text{tail}(\Sigma(a)))) \texttt{ =}$$
$$\quad\quad \texttt{split(}n, a\texttt{) in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto n :: \text{tail}(\Sigma(a)), v_2 \mapsto m :: \text{tail}(\Sigma(a)), \Sigma}(e)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v \texttt{ = size(}i, a\texttt{) in } e) =$$
$$\quad \texttt{let } v \texttt{ = } \Sigma(a)[i] \texttt{ in}$$
$$\quad \mathcal{A}^{\text{exp}}_{\Sigma}(e)$$

Figure 13. Simple transformation rules.

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v_1, \ldots, v_n \texttt{ = } f\texttt{(}a_1, \ldots, a_m\texttt{) in } e) =$$
$$\quad \texttt{let } s^1_1, \ldots, s^{\text{d}(v_1)}_1, \ldots, s^1_n, \ldots, s^{\text{d}(v_n)}_n,$$
$$\quad\quad \mathcal{V}(v_1, s_1), \ldots, \mathcal{V}(v_n, s_n) =$$
$$\quad\quad f_{\text{ext}}\texttt{(}a_1, \Sigma(a_1), \ldots, a_m, \Sigma(a_m)\texttt{) in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e)$$

Where $s_i = s^1_i :: \ldots :: s^{\text{d}(v_i)}_i$

Figure 14. Function call transformation rule

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } v_1, \ldots, v_n \texttt{ = if } a_c \texttt{ then } e_t \texttt{ else } e_f \texttt{ in } e_b) =$$
$$\quad \texttt{let } s^1_1, \ldots, s^{\text{d}(v_1)}_1, \ldots, s^1_n, \ldots, s^{\text{d}(v_n)}_n,$$
$$\quad\quad \mathcal{V}(v_1, s_1), \ldots, \mathcal{V}(v_n, s_n) \texttt{ =}$$
$$\quad\quad \texttt{if } a_c \texttt{ then } \mathcal{A}^{\text{exp}}_{\Sigma}(e_t) \texttt{ else } \mathcal{A}^{\text{exp}}_{\Sigma}(e_f) \texttt{ in}$$
$$\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e_b)$$

Where $s_i = s^1_i :: \ldots :: s^{\text{d}(v_i)}_i$

Figure 15. Branch transformation

3.4 SOAC-related Rules

The biggest problem with annotating SOACs is that, in Futhark, anonymous functions cannot be existentially quantified. Hence, before evaluating the SOAC, we must know the shape of its return value. For this section, we assume that given an anonymous function λ, it is possible to compute λ_{shape}, which returns the shape of the values that would normally be returned by λ.

Similarly to function calls, when transforming `mapT(`λ`,`a`)`, there are two possible avenues of attack.

Pre-assertion First, calculate `mapT(`λ_{shape}`,`a`)`, and assert, for each returned array, that all of its elements are identical. That is, check in advance that the `mapT` expression results in a regular array. After this check, we know with certainty the shape of the values returned by the original map computation (and that it will be regular), which we can then use to annotate the `mapT` computing the values. This rule is shown in Figure 16, generalised to handle an arbitrary amount of array arguments to `mapT`. For exposition, we use a convenience function by the name of `allEqual`, which is shown on Figure 18.

Intra-assertion Alternatively, we can compute $\lambda_{\text{shape}}(a[0])$, the shape of the first element of the result. Then, we modify λ to explicitly `assert`, for each call, that its actual return value has the same shape as that computed for the first element - we call this modified version $\lambda_{\text{checking}}$. This rule is shown in Figure 17, and is the one that has been applied on Figure 8. (For brevity we have omitted the `if`-branch guarding the case when `a` is empty.)

The former approach is only efficient if λ_{shape} is efficient compared to λ_{value} - in practice, it must not contain any loops. The latter approach is limited in that it forces shape checking inside the value computation, which means that we do not know in advance whether the shape annotation is correct. The implication is that we cannot allocate all memory required by the `mapT` in advance, which will likely prevent parallel execution on GPU, although SMP parallelism still remains possible.

However, in Section 4 we will describe an optimisation that is able to rewrite the result of the intra-assertion approach to code equivalent to the pre-assertion approach. In fact, this will just be a special case of the function call splitting described previously, combined with a method of separating assertions from the main computation. Hence, our compiler always applies the intra-assertion rule, with the expectation that a later optimisation will remove the assertions, if possible.

For `filterT`, we annotate the given lambda function, which is simple, as we know it returns just a single boolean, meaning there are no shape annotations in the return type. Hence, we pass an empty list (written as $\langle\rangle$) for the return shape annotation. One possible way to treat `filterT` is shown on Figure 19, in which we map the annotated filter function `mapT` over the original arrays, computing a boolean *flag array* that indicates which elements should be included. The number of true values of the flag array could then be counted, and used to annotate the "real" call to `filterT`. It is clear that this duplicates computation - the filter function is called twice for every element. Possible solutions include (i) a `pack` primitive that can re-use the flag array, (ii) passing the flag array as array input to the second `filterT` call then using a `mapT` with a projection to remove it from the result, or (iii) simply extending `filterT` to explicitly return the outer size of the result as a value, i.e., keep `filterT` existentially typed. The last option is particularly enticing, as we cannot in any case compute the result size of a `filterT` in advance. However, we *do* know an upper bound on the size (exactly the size of the array being filtered), which means that we are still able to allocate memory in advance. This relation is not en-

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let \{}\bar{v}^{(m)}\texttt{\} = mapT(}\lambda\texttt{, }\bar{a}^{(n)}\texttt{) in }e) = \\
&\texttt{let } ss_1^2, \ldots, ss_1^{\mathtt{d}(v_1)}, \ldots, ss_m^2, \ldots, ss_m^{\mathtt{d}(v_m)} \texttt{ =} \\
&\quad \texttt{mapT(}\lambda_{\text{shape}}\texttt{,}\bar{a}^{(n)}\texttt{) in} \\
&\texttt{let } c_1, s_1^2, \ldots, s_1^{\mathtt{d}(v_1)} \texttt{ = allEqual(}ss_1^2, \ldots, ss_1^{\mathtt{d}(v_1)}\texttt{) in} \\
&\qquad \vdots \\
&\texttt{let } c_m, s_m^2, \ldots, s_m^{\mathtt{d}(v_m)} \texttt{ = allEqual(}ss_m^2, \ldots, ss_m^{\mathtt{d}(v_m)}\texttt{) in} \\
&\texttt{let } \mathcal{V}(v_1, s_i), \ldots, \mathcal{V}(v_m, s_i) \texttt{ = <}\bar{c}^{(m)}\texttt{>mapT(}\lambda_{\text{value}}, \bar{a}^{(n)}\texttt{) in} \\
&\mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e) \\
&\qquad \text{Where } r_i = s_i^2 :: \ldots :: s_i^{\mathtt{d}(v_i)} \\
&\qquad\qquad s_i = s_i^1 :: r_i \\
&\qquad\qquad p_i = \text{tail}(\Sigma(a_i)) \\
&\qquad\qquad \lambda_{\text{shape}} = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \bar{r}^{(m)}, \bar{p}^{(n)})_{\text{shape}}.
\end{aligned}
$$

Figure 16. Map transformation rule if λ_{shape} contains no loops.

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let \{}\bar{v}^{(m)}\texttt{\} = mapT(}\lambda\texttt{, }\bar{a}^{(n)}\texttt{) in }e) = \\
&\texttt{let } s_1^2, \ldots, s_1^{\mathtt{d}(v_1)}, \ldots, s_m^2, \ldots, s_m^{\mathtt{d}(v_m)} \texttt{ =} \\
&\quad \lambda_{\text{shape}}\texttt{(}a_1\texttt{[0]}, \ldots, a_n\texttt{[0]) in} \\
&\texttt{let } \mathcal{V}(v_1, s_i), \ldots, \mathcal{V}(v_m, s_i) \texttt{ = mapT(}\lambda_{\texttt{checking}}, \bar{a}^{(n)}\texttt{) in} \\
&\mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e) \\
&\qquad \text{Where } r_i = s_i^2 :: \ldots :: s_i^{\mathtt{d}(v_i)} \\
&\qquad\qquad s_i = s_i^1 :: r_i \\
&\qquad\qquad p_i = \text{tail}(\Sigma(a_i)) \\
&\qquad\qquad \lambda_{\text{shape}} = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \bar{r}^{(m)}, \bar{p}^{(n)})_{\text{shape}}.
\end{aligned}
$$

Figure 17. Map transformation rule if λ_{shape} is contains loops.

```
allEqual(ss_1, ..., ss_n) =
  let oks_1 = mapT(op ==, ss_1, rotate(1,ss_1)) in
  let ok_1 = reduceT(op &&, oks_1) in
      ⋮
  let oks_n = mapT(op ==, ss_n, rotate(1,ss_n)) in
  let ok_n = reduceT(op &&, oks_n) in
  {assert(ok_1 && ... && ok_n), ss_1[0], ..., ss_n[0]}
```

Figure 18. The `allEqual` convenience function.

coded in the type system, but it is not difficult for a code generator to discover and exploit it. This approach is shown in Figure 20.

The transformation rule for `reduceT` is also shown on Figure 21. We exploit the invariant that the function may not change the shape of the accumulator, and that the return value of the reduction function must hence have the same shape as the initial value of the accumulator. In the practical implementation, we also extend the reduction function to explicitly `assert` that its result is indeed the shape that is expected, but this has been elided for brevity. Also, in contrast to `filterT` and `mapT`, not all of the $2n$ parameters to the

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } \bar{v}^{(n)} \texttt{ = filterT(}\lambda\texttt{, }\bar{a}^{(n)}\texttt{) in }e) = \\
&\quad \texttt{let [bool, } \Sigma(a_1)\texttt{] } ok \texttt{ = mapT(}\lambda'\texttt{, }\bar{a}^{(n)}\texttt{) in} \\
&\quad \texttt{let } s_o \texttt{ = count(}ok\texttt{) in} \\
&\quad \texttt{let } \mathcal{V}(v_1, s_1), \ldots, \mathcal{V}(v_n, s_n) \texttt{ = filterT(}\lambda', \bar{a}^{(n)}\texttt{) in} \\
&\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e) \\
&\qquad \text{Where } s_i = s_o :: \text{tail}(\Sigma(a_i)) \\
&\qquad\qquad \lambda' = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \langle\rangle, \text{tail}(\Sigma(a_1)), \cdots, \text{tail}(\Sigma(a_n)))_{\text{value}}
\end{aligned}
$$

Figure 19. `filterT` duplicating work.

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let } \bar{v}^{(n)} \texttt{ = filterT(}\lambda\texttt{, }\bar{a}^{(n)}\texttt{) in }e) = \\
&\quad \texttt{let } s_o, \mathcal{V}(v_1, s_1), \ldots, \mathcal{V}(v_n, s_n) \texttt{ = filterT(}\lambda', \bar{a}^{(n)}\texttt{) in} \\
&\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto s_1, \ldots, v_n \mapsto s_n, \Sigma}(e) \\
&\qquad \text{Where } s_i = s_o :: \text{tail}(\Sigma(a_i)) \\
&\qquad\qquad \lambda' = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \langle\rangle, \text{tail}(\Sigma(a_1)), \cdots, \text{tail}(\Sigma(a_n)))_{\text{value}}
\end{aligned}
$$

Figure 20. `filterT` without duplicating work.

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let \{}\bar{v}^{(n)}\texttt{\} = reduceT(}\lambda\texttt{, \{}\bar{e}^{(n)}\texttt{\}, }\bar{a}^{(n)}\texttt{) in }e) = \\
&\quad \texttt{let } \mathcal{V}(v_1, \Sigma(e_1)), \ldots, \mathcal{V}(v_n, \Sigma(e_n)) \texttt{ =} \\
&\quad \texttt{reduceT(}\mathcal{A}^{\texttt{lam}}_{\Sigma}(\lambda, \bar{r}^{(n)}, \bar{p}^{(2n)})_{\text{value}}\texttt{, } \bar{e}^{(n)}\texttt{, } \bar{a}^{(n)}\texttt{) in} \\
&\quad \mathcal{A}^{\text{exp}}_{v_1 \mapsto \Sigma(e_1), \ldots, v_n \mapsto \Sigma(e_n), \Sigma}(e) \\
&\qquad \text{Where } r_i = \Sigma(e_i) \\
&\qquad\qquad p_i = \begin{cases} \Sigma(e_i) & \text{if } i \leq n \\ \text{tail}(\Sigma(a_i)) & \text{if } i > n \end{cases} \\
&\qquad\qquad \lambda_{\text{value}} = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \langle\rangle, \text{tail}(\Sigma(a_1)), \cdots, \text{tail}(\Sigma(a_n)))_{\text{value}}
\end{aligned}
$$

Figure 21. The `reduceT` transformation rule (`scanT` is similar).

reduction function are array elements - the first n are the accumulator - so we need to make sure we pick out the right annotations.

4. Shape Optimisation

The rules presented previously are naive in the way that they sometimes assume that the shape of expressions are more existential than they need to be,[2] resulting in superfluous bindings. This is illustrated in Figure 22. While the first `if`-expression is indeed existentially typed by necessity, as the shape of the result of the second branch (`q`) is not free outside the branch, both branches of the second `if`-expression return the same shape (`p`). Hence, we can remove the shape binding `m`, obtaining the code in Figure 23.

We perform a similar optimisation for calls to existentially typed functions. For every call to f_{ext}, we check whether f_{shape} is "efficient", i.e., does not duplicate much of the work of f_{val}. If so, we replace the call to f_{ext} with separate calls to f_{shape} and f_{value}, with the result that existential bindings disappear. Of course, this does not necessarily mean that we gain much in terms of ability to pre-allocate memory, unless f_{shape} can be hoisted out of inner loops (probably after inlining). Importantly, however, it also means that the `assert`-expressions arising from the application of the

[2] This may seem strange from a type theory perspective - in the compiler, the "existentiality" of some expressions is given by an annotation in the syntax tree node.

```
let int n, [int,n] a =
  if c1 then let [int,p] a1 = iota(p) in
              p, a1
         else let q = f(p) in
              let [int,q] a2 = iota(q) in
              q, a2
let int m, [int,m] b =
  if c2 then let [int,p] b1 = replicate(p,x) in
              p, b1
         else let [int,p] b2 = replicate(p,y) in
              p, b2
...
```

Figure 22. Existentially typed branches.

```
let int n, [int,n] a =
  if c1 then let [int,p] a1 = iota(p) in
              p, a1
         else let q = f(p) in
              let [int,q] a2 = iota(q) in
              q, a2
let [int,p] b =
  if c2 then let [int,p] b1 = replicate(p,x) in
              b1
         else let [int,p] b2 = replicate(p,y) in
              b2
...
```

Figure 23. Shape-optimised branches.

`mapT` transformation rule from Figure 17 can be separated from the main loop and optimised separately, using the same technique that the compiler employs to optimise bounds checks [23], which was also demonstrated in Figure 10.

5. Related Work

A large body of related work is the work on libraries and embedded domain specific languages for programming massively parallel architectures, such as GPGPUs. Initial examples of such libraries include Nikola [27], a Haskell library for targeting Nvidia CUDA GPUs. Later work includes Accelerate [8] and Obsidian [10], Haskell libraries that, with different sets of fusion- and optimization techniques, targets OpenCL and CUDA.

Most related to this work is the work on SaC [16, 17], which seeks to provide a common ground between functional and imperative domains for targeting parallel architectures, including both multi-processor architectures [14] and massively data-parallel architectures [18]. SaC uses `with` and `for` loops to express map-reduce style parallelism and sequential computation, respectively. More complex array constructs can be compiled into `with` and `for` loops, as demonstrated, for instance, by the compilation of the APL programming language [24] into SaC [15]. A dependent-type method for verifying array related invariants is presented in a simplified offspring of SaC, named Qube [41]. Compared to SaC, Futhark holds on to the SOAC combinators also in the intermediate representations in order to perform critical optimizations, such as fusion, even in cases involving filtering and scans, which are not straightforward constructs for SaC to cope with.

Also related to the present work is the work on array languages in general (including APL [24] and its derivatives) and the work on capturing the essential mathematical algebraic aspects of array programming [19]. Compilers for array languages also depend on inferring shape information either dynamically or statically, although they can often assume that the arrays operated on are regular, which is not the case for Futhark programs. Another piece of related work is the work on the FISh [25] programming language, which uses partial evaluation and program specialization for resolving shape information at compile time.

Much work has also gone into investigating expressive type systems, based on dependent types, which allow for expressing more accurately, the assumptions of higher-order operators for array operations [38, 40, 41]. Compared to the present work, such type systems may give the programmer certainty about particular execution strategies implemented by a backend compiler. The expressiveness, however, comes at a price. Advanced dependent type systems are often very difficult to program and modularity and reusability of library routines require the end programmer to grasp, in detail, the underlying, often complicated, type system. Computer algebra systems [43, 44] have also provided for a long time a compelling application of dependent types in order to express accurately the rich mathematical structure of applications, but inter-operating across such systems remains a significant challenge [9, 31].

A somewhat orthogonal approach has been to extend the language operators such that size and bounds checking invariants always hold [12], the downfall being that non-affine indexing might appear. The Futhark strategy is instead to rely on advanced program analysis and compilation techniques to implement a pay-as-you-go scheme for programming massively parallel architectures.

Another strand of related work is the work on effect systems for region-based memory management [39], in particular, the work on multiplicity inference and region representation analysis in terms of physical-size inference [3, 42]. Whereas the goal of multiplicity inference is to determine an upper bound to the number of objects stored into a region at runtime, physical-size inference seeks to compute an upper bound to the number of bytes stored in a single write. Compared to the present work, multiplicity inference and physical-size inference are engineered to work well for common objects such as pairs and closures, but the techniques work less well with objects whose sizes are determined dynamically.

Much related work has been carried out in the area of supporting nested parallelism, including the seminal work on flattening of nested parallelism in NESL [4, 5] and in the more recent work on flattening [2, 45]. Investigating the use of a flattening transform for supporting parallel implementations of arbitrarily nested parallelism in Futhark is future work.

6. Conclusions

In this paper we have introduced the notion of size slicing for separating the computation of array-like object sizes from the computation of the values to be stored in those objects. The size slicing scheme is presented as a set of transformation rules, which are guaranteed to incur only a constant overhead, which can be further effectively optimised via traditional optimisations such as inlining, fusion, constant propagation, common-subexpression elimination and dead-code elimination.

The main motivation for the present work have been to pave the road for generating code for massively parallel architectures, such as GPGPU kernels, where, typically, the memory for the argument and intermediate data is statically pre-allocated. By combining size slicing with traditional hoisting techniques, we believe this to be possible.

Acknowledgments

The authors would like to thank the reviewers for the constructive comments. This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information

Technology' (http://hiperfit.dk) under contract number 10-092299.

References

[1] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*, volume 761 of *LNCS*, pages 41–51, 1993.

[2] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 247–258, Sept. 2012.

[3] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *ACM Symposium on Principles of Programming Languages*, POPL'96, pages 171–183. ACM Press, January 1996.

[4] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[5] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-02313-X.

[6] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[7] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Procs. Langs. Comp. Parallel Computing (LCPC)*, pages 141–154. Springer-Verlag, 1994.

[8] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *International Workshop on Declarative Aspects of Multicore Programming*, DAMP'11, pages 3–14, 2011.

[9] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Procs. Int. Symp. Symbolic and Numeric Alg. for Scientific Computing (SYNASC)*, pages 119–130. Mirton Publishing House, 2004.

[10] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *International Workshop on Declarative Aspects of Multicore Programming*, DAMP'12, pages 21–30, 2012.

[11] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Int. Par. and Distr. Processing Symp. (PDPS)*, pages 20–29, 2002.

[12] M. Elsman and M. Dybdal. Compiling a Subset of APL Into a Typed Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[13] K. Fraser and T. Harris. Concurrent Programming Without Locks. *Trans. of Comput. Syst. (TOCS)*, 25(2), May 2007.

[14] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming (JFP)*, 15(3): 353–401, 2005.

[15] C. Grelck and S.-B. Scholz. Accelerating APL programs with SAC. In *Proceedings of the Conference on APL '99: On Track to the 21st Century*, APL'99, pages 50–57. ACM, 1999.

[16] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[17] C. Grelck and F. Tang. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

[18] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*, pages 15–24. ACM, 2011.

[19] G. Hains and L. M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 1993.

[20] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[21] T. Henriksen. Exploiting functional invariants to optimise parallelism: a dataflow approach. Master's thesis, DIKU, Denmark, 2014.

[22] T. Henriksen and C. E. Oancea. A T2 Graph-Reduction Approach to Fusion. In *Procs. Funct. High-Perf. Comp. (FHPC)*, pages 47–58. ACM, 2013. ISBN 978-1-4503-2381-9.

[23] T. Henriksen and C. E. Oancea. Bounds Checking: An Instance of Hybrid Analysis. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[24] K. E. Iverson. *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[25] C. B. Jay. Programming in fish. *International Journal on Software Tools for Technology Transfer*, 2(3):307–315, 1999.

[26] K. Kennedy, C. Koelbel, and H. Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Procs. Conf. on History of Prog. Lang. (HOPL III)*, pages 7–1–7–22. ACM, 2007.

[27] G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the 3rd ACM International Symposium on Haskell*, pages 67–78, 2010.

[28] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial Software on GPUs: between Haskell and Fortran. In *Funct. High-Perf. Comp. (FHPC'12)*, 2012.

[29] C. E. Oancea and A. Mycroft. Set-Congruence Dynamic Analysis for Software Thread-Level Speculation. In *Procs. Langs. Comp. Parallel Computing (LCPC)*, pages 156–171, 2008.

[30] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, pages 509–520, 2012.

[31] C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface between Two Approaches to Computer Algebra. In *Procs. Int. Symp. Symbolic Alg. Comp. (ISSAC)*, pages 261–269. ACM, 2005.

[32] C. E. Oancea, A. Mycroft, and S. M. Watt. A New Approach to Parallelising Tracing Algorithms. In *Procs. Int. Symp. on Memory Management (ISMM)*, pages 10–19. ACM, 2009. .

[33] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *Procs. Sym. Principles of Prog. Lang. (POPL)*, pages 549–562. ACM, 2011.

[34] P. Rundberg and P. Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocs. *Journal of Instruction-Level Parallelism*, 1999.

[35] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan. 1992. ISSN 1045-3563.

[36] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12 (3):66–73, 2010. ISSN 0740-7475.

[37] M. M. Strout, L. Carter, and J. Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Procs. Int. Conf. Prog. Lang. Design and Implem. (PLDI)*, pages 91–102. ACM, 2003.

[38] P. Thiemann and M. M. T. Chakravarty. Agda meets accelerate. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, IFL'2012, 2013. Revised Papers, Springer-Verlag, LNCS 8241.

[39] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation (HOSC)*, 17(3):245–265, September 2004.

[40] K. Trojahner and C. 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).

[41] K. Trojahner and C. Grelck. Descriptor-free representation of arrays with dependent types. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 100–117. Springer-Verlag, 2011.

[42] M. Vejlstrup. Multiplicity inference. Master's thesis, Department of Computer Science, University of Copenhagen, September 1994.

[43] S. M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Handbook of Computer Algebra*, pages 154–160, 2003.

[44] S. M. Watt, R. D. Jenks, R. S. Sutor, and B. M. Trager. The Scratchpad II Type System: Domains and Subdomains. In *Procs of Computing Tools For Scientific Problem Solving*, pages 63–82. A. Miola ed. Academic Press, 1990.

[45] Y. Zhang and F. Mueller. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP'12, pages 340–349, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4796-1.