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@sigkilo.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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Submitted to FHPC’14, September 4, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3040-4/14/09 ...$15.00.
http://dx.doi.org/10.1145/2636228.2636238
an array produced by \texttt{map} would be known only after the execution of \texttt{map}, that is, the array would not be amenable to static allocation.

We optimize the common case via a \textit{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 \texttt{map}, \texttt{1 o op}, or recursive functions appear in the shape slice.

Related dependent-type systems \cite{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 \cite{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., “Fuphark”) enables a regular notion of nested parallelism via a set of six second-order array combinators (SOACs): \texttt{map}, \texttt{reduce}, \texttt{filter}, \texttt{scan}, \texttt{multireduce} (not implemented yet), and \texttt{redemap} (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 \cite{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 \cite{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, and hoisting of invariant terms out of recurrences in loops and \textit{soacs}. The simplification rules are critical for optimising the result of higher-level optimisations, such as producer-consumer fusion \cite{21, 22}, bounds-checking analysis \cite{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 \(\bar{z} = z_0, \ldots, z_{n-1}\) we often write it on the form \(\bar{z}^{(n)}\) and we write \(\bar{z}, \bar{x}\) to denote the sequence \(z, z_0, \ldots, z_{n-1}\). We will write \(d(\tau)\) to indicate the rank (number of dimensions) of a type \(\tau\). We may also write \(d(v)\), where \(v\) is not itself a type, but something which \textit{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 \(x, y, \text{ and } s\), and we use \(g\) and \(h\) to range over function names.
- A variable may have a scalar type (i.e., \texttt{bool}, \texttt{int}, \texttt{real}), or a multidimensional (regular) array type, such as \([\text{real}]\), 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 \texttt{mapT}, \texttt{reduceT}, \texttt{filterT}, and so on. Moreover, named functions may only be defined at toplevel. Named functions may be mutually recursive. Expressions in the intermediate language are normalized. That is, they consist of a sequence of \texttt{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 \texttt{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
and zip a composition between SOAC set of the intermediate language (e.g., mapT, scanT, filterT) types and type schemes to and from their extended counter parts. A the treatment of operators and we shall be implicit about converting

\[ \rho : \rho_1 \to \rho_2 \] (fun/lambda type)

\[ p := \tau_1 x_1, \ldots, \tau_n x_n \] (n-array pattern)

\[ e := \langle x_1, \ldots, x_n \rangle \] (tuple exp)

\[ \text{let } p = ct \in e \] (constant binding)

\[ \text{let } p = \{ x_1, \ldots, x_n \} \in e \] (array indexing)

\[ \text{let } p = \langle x_1, \ldots, x_n \rangle \in e \] (variable renaming)

\[ \text{let } p = \text{op}(\langle \rangle) \in e \] (operator call)

\[ \text{let } p = \text{eq}(\langle \rangle) \in e \] (function-call)

\[ \text{let } p = \langle \rangle \text{ if } s \text{ then } e_1 \text{ else } e_2 \] (if binding)

\[ \text{let } q = \text{loop}(\langle \rangle) \] (do-loop binding)

\[ \text{for } s < S \text{ do } e_1 \] (for in e2)

\[ \text{let } a = x \] (simple argument)

\[ \text{fn } \rho(p) \Rightarrow e \] (function argument)

\[ P := \text{fun } \rho g(p) = e; P \] (named function definition)

\[ e \] (main expression)

![Figure 2. Source, Size-agnostic IR for Futhark.](image)

loop has the semantics of the tail-recursive call:

\[ \text{let } p = h(\{0, S, x_1, \ldots, x_n\}) \in e_2, \text{ where } h \text{ is defined as:} \]

\[ \text{fun } \{\tau_1, \ldots, \tau_n\} \to h(\{\text{int } s, \text{ int } S, \tau_1 p_1, \ldots, \tau_n p_n\}) = \]

\[ \text{if } s \not> S \text{ then } \{\tau_1, \ldots, \tau_n\} \text{ else } h(s+1, S, e_1) \]

• 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:

\[ \langle \tau \rangle \triangleq \forall \alpha. \langle \tau \rangle \]

Extended types (\(\langle \rangle\)) and extended type schemes (\(\langle \rangle\)) 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 \(\langle \rangle\) is an instance of an extended type scheme \(\forall \alpha. \langle \rangle\) written \(\langle \rangle \geq \langle \rangle\), if there exists a substitution \(S\) such that \(S(\langle \rangle) \geq \langle \rangle\).

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., \texttt{mapT}) are the tuple-of-array version of the user language SOACS (e.g., \texttt{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 SOACC \(\texttt{mapT, scanT, filterT}\) operator can be described as a composition between \texttt{unzip}, the user-language SOAC (e.g., \texttt{map}), and \texttt{zip}, where the unnamed function is suitably modified to work with the flat sequence of array arguments.

\[ \text{op}(\langle \rangle) \] Description

\[ \emptyset s \] Unary scalar operation on \(s\).

\[ s_1 \oplus s_2 \] Binary scalar operation on \(s_1\) and \(s_2\).

\[ \text{size}(s,x) \] Returns the size of dim \(s\) of \(x\).

\[ \text{iota}(x) \] Returns the vector \([0, \ldots, x-1]\).

\[ \text{replicate}(s,x) \] Returns an array of rank one higher than \(x\)'s rank, containing an \(s\)-times replication of \(x\).

\[ \text{split}(s,x) \] Returns a pair of arrays resulting from splitting array \(x\) at index \(s\).

\[ \text{concat}(x_1,x_2) \] Returns the array resulting from concatenating the two arrays \(x_1\) and \(x_2\) along the last dimension.

\[ \text{mapT}(\lambda, \vec{x}) \] Equivalent to \((\text{unzip} \circ \text{map' \circ} \text{zip})\vec{x}, \]

\[ \text{mapT}(\lambda, \vec{x}) = (\text{unzip} \circ \text{map' \circ} \text{zip})\vec{x}, \]

\[ \text{reduceT}(\lambda, \vec{x}) \] Equivalent to \((\text{reduce}'(\vec{x}) \circ \text{zip})\vec{x}, \]

\[ \text{filterT}(\lambda, \vec{x}) \] Equivalent to \((\text{unzip} \circ \text{filter'} \circ \text{zip})\vec{x} \]

Figure 3. Description of SOAC operators.

\[ \text{op} \] TySchr(op)

\[ \text{size} \] :: \(\text{Oa.}\{\{\text{int } \alpha\} \to \text{int}\]

\[ \text{iota} \] :: \(\text{int} \to \{\text{int}\]

\[ \text{split} \] :: \(\text{Oa.}\{\{\text{int } \alpha\} \to \{\text{int}\]

\[ \text{concat} \] :: \(\text{Oa.}\{\{\text{int}\}\to \{\text{int}\]

\[ \text{mapT} \] :: \(\forall \alpha.\{\{\text{int}\}\to \{\text{int}\]

\[ \text{reduceT} \] :: \(\forall \alpha.\{\{\text{int}\}\to \{\text{int}\]

\[ \text{filterT} \] :: \(\forall \alpha.\{\{\text{int}\}\to \{\text{int}\]

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

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

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

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:

• \text{iota}, \text{replicate, and split} assume a positive first argument.
\[ \tau ::= t \mid [\tau, s] \quad \text{(outer array of size } s) \]

\[ \phi ::= \forall s_1, (\tau_1) \rightarrow \exists s_2, \tau_2 \quad \text{(sizes of args } s \in s_1, \text{ results } \in s_2) \]

\( \text{concat} \) ::= \( \forall s_1, s_2, \tau \mid [\tau_1, \tau_2] \rightarrow [\tau_1, s + s_2] \)

\( \text{reduce} \) ::= \( \forall s \, \alpha(n) \cdot \{\phi, \alpha(n), \{\alpha_1, s_1, \ldots, \alpha_n, s_n\} \rightarrow \{\alpha(n)\} \}
\]

where \( \phi = \{\alpha(n), \alpha(n)\} \rightarrow \{\alpha(n)\} \)

\( \text{filter} \) ::= \( \forall s_1, \alpha(n) \cdot \{\phi, \{\alpha_1, s_1, \ldots, \alpha_n, s_n\}\}
\]

\( \rightarrow \exists s_2, \{\alpha_1, s_2, \ldots, \alpha_n, s_2\} \)

where \( \phi = \{\alpha(n)\} \rightarrow \text{bool} \) and \( 0 \leq s_2 \leq s_1 \)

\( \text{map} \) ::= \( \forall \{\tau(n+m)\} \cdot \{\alpha(n) \rightarrow \beta(m)\}, \{\alpha_1, s_1, \ldots, \alpha_n, s_n\} \rightarrow \{\beta_1, s_1, \ldots, \beta_m, s_m\} \),

where \( \beta_i = \{s_{i+1}^\phi, s_{i+1}^\phi, \ldots, s_{i+1}^\phi \}, \text{ and } s_i^\phi, s_i^\phi \in s_j^\phi, \forall i \in \{1 \ldots m\}, j \in \{1 \ldots k\}. \text{ Similar for } \alpha. \)

**Figure 6.** Dependent-size types for various SOACs.

- **map** is guaranteed to receive arguments of identical outermost size, which also matches the outermost size of all result arrays. However, map assumes that its function argument produces arrays of identical shape for each element of the input array.

- **filter** 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).

- **reduce** and **scan** receive arguments of identical outermost size, and **scan** results have outermost size equal to that of the input. The semantics for reduce and scan 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 map 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 \( \tau(n+m) \) and outermost size \( s \), and (ii) the existentially quantified inner-dimension sizes \( s_i \). 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.

\footnote{In the user language \texttt{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], 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] S - concat(S, P) in
  R

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

fun [int] f_shape(int N, int M, [[real,M],N] a) =
  let [real] S, [real] P =
    mapT( fn ([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,N] S, [real,N] P =
    let [real] S, [real] P =
      for each array argument, e.g., since array $s$ has been inferred to be a two-dimensional array of reals - as a result of a map with a function of signature $[[\text{real}]] \rightarrow \text{real}$ - but its shape is not known yet.

The first stage, demonstrated in Figure 8, transforms the program into an optimised 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 $\text{mapT}$ to $a$ will result in arrays $S$ and $P$ of (outermost) size $N$, and as such, $\text{concat}(S,P)$ 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 $\{\text{int}, [\text{real}, ?0]\}$ denoting that the size of the array result is the first $\text{int}$ 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 $f$ is now $(k, R)$ where $k$ is the size of $R$, and the result type of $f$ is $\{\text{int}, [\text{real}, ?0]\}$ denoting that the size of the array result is the first $\text{int}$ result, and finally,
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 N+N in
  let c = assert(reduce(&&, True, bs)) in
  let int s = if N == 0 then assert(True) else N+N in
  let S = if N == 0 then 0 else N+N in
  in B[s]

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
  in B[s]

C. Rewriting mapT to replicate...

let p = \( s \map_{\text{value}}(S, N, M, a) \rightarrow \) f_value(S, N, M, a)

D. Simplifying reduction...

let c = assert(reduce(&&, True, bs)) in

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 re-
cursive, 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 pro-
gram, where all types (except return types of top-level functions)
have full shape annotations, and we explicitly check the regular-
arity 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 vari-
ables 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 exis-
tential function \( f_{\text{ext}} \), 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 \( \{ \tau_1, \ldots, \tau_n \} \),
then the return type of \( f_{\text{ext}} \) is \( \{ \text{int } \in \{ \tau_1, \ldots, \tau_n \} \} \),
where \( d(\tau) \) is the rank of \( \tau \), and \( \tau' \) is \( \tau \) shape-annotated with
\( \text{int } d(\tau) \). For example, if \( f \) returns type \( \{ [\text{int}] \} \), \( f_{\text{ext}} \) will return
return type \( \{ \text{int}, \text{int }, [\text{int}, ?1], \} \). Here, 70 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
\( [\text{int}][p] \), then \( f_{\text{ext}} \) will take three parameters int n, int m, and
\( [\text{int}, m], n \).

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

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

as actual values being passed around in the program.
We assume we have an existential function

\[ \mathcal{T}(\tau, s^{(d(\tau))}) = [[[\tau, s_{\theta}(\tau)], \cdots], s_1] \]

\[ \mathcal{T}(t, s^{(d(t))}) = t \]

\[ \mathcal{V}(\tau, x, s^{(d(\tau))}) = \mathcal{T}(\tau, s^{(d(\tau))}) x \]

Where \( t \) is the basic type of \( \tau \) and \( d(\tau) \) is the rank of \( \tau \).

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 \( x \) with the shapes in \( s \), which is a sequence of variable names whose length is equal to the rank of \( \tau \). Similarly, \( \mathcal{T}(\tau, s) \) performs this annotation on a type. As an example,

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

and

\[ \mathcal{V}([[\text{int}], a, n, m] = [[[\text{int}], m], n, a] \]

3.1.1 Size Slicing

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

\[
\begin{align*}
\text{fun} \ (\int^{(r_1)} & \cdots, \int^{(n)}_{\text{ext}}, \tau_1', \cdots, \tau_n') \ f_{\text{ext}}(p^n) = \\
& \text{let bnds in} \\
& \{ \pi^{(r_1)}, \cdots, \pi^{(r_n)}, \pi^n \}
\end{align*}
\]

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

\[
\begin{align*}
\text{fun} \ (\int^{(r_1)} & \cdots, \int^{(n)}_{\text{ext}}, \tau_1', \cdots, \tau_n') \ f_{\text{shape}}(p^n) = \\
& \text{let bnds in} \\
& \{ \pi^{(r_1)}, \cdots, \pi^{(r_n)}, \pi^n \}
\end{align*}
\]

For the value function \( f_{\text{value}} \), 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 \( f_{\text{shape}} \), 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 \( s' \)s of \( f_{\text{value}} \) are instantiated with the results of \( f_{\text{shape}} \):

\[
\begin{align*}
\text{fun} \ (\tau_1, \pi^{(r_1)}), \cdots, (\tau_n, \pi^{(r_n)}) & \ f_{\text{value}}(\pi^{(r_1)}, \cdots, \pi^{(r_n)}, p^n) = \\
& \text{let bnds in} \\
& \{ \pi^n \}
\end{align*}
\]

Usually, there will be a large amount of dead and unnecessary code in \( f_{\text{shape}} \), 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 \( A_{\Sigma}^{\text{ext}}(b) \) computes the annotated version of the body \( b \) in the environment \( \Sigma \) and returns shapes as well as values (that is, its type will contain existentials). The environment \( \Sigma \) 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 \( x :: x.s \).

The function \( A_{\Sigma}^{\text{val}}(f) \) computes the existentially-quantified function \( f_{\text{ext}} \), by using \( A_{\Sigma}^{\text{ext}} \) 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 \( A_{\Sigma}^{\text{ext}}(\lambda, \tau, \varphi) \). This is similar to the transformation transformation, except that (i) we work within an environment \( \Sigma \), and (ii) we know in advance the intended shape of the result \( \tau \) and parameters \( \varphi \), 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 \( A_{\Sigma}^{\text{ext}}(b) \).

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:

\[
A_{\Sigma}^{\text{ext}}(\langle x^{(n)} \rangle) = \langle \pi^{(r_1)}, \cdots, \pi^{(r_n)}, x_1, \cdots, x_n \rangle
\]

where \( \pi^{(r_1)} = \Sigma(x_1) \)

\[
\cdots
\]

\[
\pi^{(r_n)} = \Sigma(x_n)
\]

Other simple cases are listed in Figure 13. Note that, for \( \text{concat} \) and \( \text{split} \), we introduce new bindings in the program to record the outer sizes of the resulting arrays. Also, observe how \( \text{size} \) 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 \( f_{\text{ext}} \), 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 \( f_{\text{shape}} \) exists for \( f_{\text{ext}} \).

Annotating an \( \text{if-then-else} \) 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.
\[ A_{\Sigma}^{\text{exp}}(\text{let } v = \text{iota}(n) \text{ in } e) = \]
\[ \text{let } [\text{int}, n] \ v = \text{iota}(n) \text{ in } A_{\text{v} = n : \Sigma}(e) \]
\[ A_{\Sigma}^{\text{exp}}(\text{let } v = \text{replicate}(n, a) \text{ in } e) = \]
\[ \text{let } V(v, n :: \Sigma(a)) = \text{iota}(a) \text{ in } A_{\text{v} = n : \Sigma(a), \Sigma}(e) \]
\[ A_{\Sigma}^{\text{exp}}(\text{let } v = \text{concat}(a_1, a_2) \text{ in } e) = \]
\[ \text{let } n = \text{head}(\Sigma(a_1)) + \text{head}(\Sigma(a_2)) \text{ in } \]
\[ \text{let } V(v, n :: \text{tail}(\Sigma(a_1))) = \text{concat}(a_1, a_2) \text{ in } A_{\text{v} = n : \text{tail}(\Sigma(a_1)), \Sigma}(e) \]
\[ A_{\Sigma}^{\text{exp}}(\text{let } v_1, v_2 = \text{split}(n, a) \text{ in } e) = \]
\[ \text{let } n = \text{head}(\Sigma(a)) - n \text{ in } \]
\[ \text{let } V(v_1, n :: \text{tail}(\Sigma(a))), V(v_2, (m, \text{tail}(\Sigma(a)))) = \]
\[ \text{split}(n, a) \text{ in } A_{\text{v}_1 = n : \text{tail}(\Sigma(a))}, v_2 = m : \text{tail}(\Sigma(a)), \Sigma}(e) \]
\[ A_{\Sigma}^{\text{exp}}(\text{let } v = \text{size}(i, a) \text{ in } e) = \]
\[ \text{let } v = \Sigma(a)[i] \text{ in } A_{\Sigma}(e) \]

Figure 13. Simple transformation rules.

\[ A_{\Sigma}^{\text{exp}}(\text{let } v_1, \ldots, v_n = f(a_1, \ldots, a_m) \text{ in } e) = \]
\[ \text{let } s_1, \ldots, s_1^{d(v_1)}, s_n, \ldots, s_n^{d(v_n)}, \]
\[ V(v_1), \ldots, V(v_n), s_n = \]
\[ f_{\text{ext}}(a_1, \Sigma(a_1), \ldots, a_m, \Sigma(a_m)) \text{ in } A_{v_1 = s_1, \ldots, v_n = s_n, \Sigma}(e) \]

Where \( s_1 = s_1^1 : \ldots : s_1^{d(v_1)} \)

Figure 14. Function call transformation rule

\[ A_{\Sigma}^{\text{exp}}(\text{let } v_1, \ldots, v_n = \text{if } a \text{ then } e_1 \text{ else } e_2 \text{ in } e_3) = \]
\[ \text{let } s_1, \ldots, s_1^{d(v_1)}, s_n, \ldots, s_n^{d(v_n)}, \]
\[ V(v_1), \ldots, V(v_n), s_n = \]
\[ \text{if } a \text{ then } A_{\Sigma}(e_1) \text{ else } A_{\Sigma}(e_2) \text{ in } A_{v_1 = s_1, \ldots, v_n = s_n, \Sigma}(e_3) \]

Where \( s_1 = s_1^1 : \ldots : s_1^{d(v_1)} \)

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 \( \lambda \), it is possible to compute \( \lambda_{\text{shape}} \), which returns the shape of the values that would normally be returned by \( \lambda \).

Similarly to function calls, when transforming \( \text{mapT}(\lambda, a) \), there are two possible avenues of attack.

**Pre-assertion** First, calculate \( \text{mapT}(\lambda_{\text{shape}}(a)) \), and assert, for each returned array, that all of its elements are identical. That is, check in advance that the \( \text{mapT} \) 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 \( \text{mapT} \) computing the values. This rule is shown in Figure 16, generalised to handle an arbitrary amount of array arguments to \( \text{mapT} \). For exposition, we use a convenience function by the name of \( \text{allEqual} \), 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 \( \lambda \) 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 \( \lambda_{\text{shape}} \) is efficient compared to \( \lambda_{\text{value}} \) - 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 \( \text{mapT} \) 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 \( \text{filterT} \), 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 \( () \) for the return shape annotation. One possible way to treat \( \text{filterT} \) is shown on Figure 19, in which we map the annotated filter function \( \text{mapT} \) 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 \( \text{filterT} \). It is clear that this duplicates computation - the filter function is called twice for every element. Possible solutions include (i) a \text{pack} primitive that can reuse the flag array, (ii) passing the flag array as array input to the \text{filterT} call then using a \text{mapT} with a projection to remove it from the result, or (iii) simply extending \( \text{filterT} \) to explicitly return the outer size of the result as a value, i.e., keep \( \text{filterT} \) existentially typed. The last option is particularly enticing, as we cannot in any case compute the result size of a \( \text{filterT} \) 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-
\( A^\text{exp}\) (let \( \{ e \} \) = mapT(\( \lambda \), \( \bar{a}^{(n)} \)) in \( e \)) =  

let \( s_1^1, \ldots, s_1^{d(v_1)}, \ldots, s_n^1, \ldots, s_n^{d(v_n)} = \) mapT(\( \lambda_{\text{shape}}, \bar{a}^{(n)} \)) in  

let \( c_1, s_1^1, \ldots, s_1^{d(v_1)} = \text{allEqual}(s_1^1, \ldots, s_1^{d(v_1)}) \) in  

\vdots  

let \( c_m, s_m^1, \ldots, s_m^{d(v_m)} = \text{allEqual}(s_m^1, \ldots, s_m^{d(v_m)}) \) in  

let \( \forall(v_1, s_1), \ldots, \forall(v_n, s_n) = \langle \forall^{(n)} \rangle \) mapT(\( \lambda_{\text{value}}, \bar{a}^{(n)} \)) in  

\( A^\text{exp}\) (let \( \bar{v}^{(n)} = \text{filterT}(\lambda, \bar{a}^{(n)}) \) in \( e \)) =  

let \[ \text{bool, } \Sigma(a_1) \] ok = mapT(\( \lambda' \), \( \bar{a}^{(n)} \)) in  

let \( s_o = \text{count}(\text{ok}) \) in  

let \( \forall(v_1, s_1), \ldots, \forall(v_n, s_n) = \text{filterT}(\lambda', \bar{a}^{(n)}) \) in  

where \( s_i = s_o :: \text{tail}(\Sigma(a_1)) \)  

\( \lambda' = A^\text{lam}_\Sigma(\lambda, j, \text{tail}(\Sigma(a_1)), \ldots, \text{tail}(\Sigma(a_n)))\) value

**Figure 19.** \( \text{filterT} \) duplicating work.

\( A^\text{exp}\) (let \( \bar{v}^{(n)} = \text{filterT}(\lambda, \bar{a}^{(n)}) \) in \( e \)) =  

let \( s_o, \forall(v_1, s_1), \ldots, \forall(v_n, s_n) = \text{filterT}(\lambda', \bar{a}^{(n)}) \) in  

\( A^\text{exp}\) (let \( \bar{v}^{(n)} = \text{filterT}(\lambda, \bar{a}^{(n)}) \) in \( e \)) =  

\( \forall(v_1, s_1), \ldots, \forall(v_n, s_n) = \text{filterT}(\lambda', \bar{a}^{(n)}) \) in  

where \( s_i = s_o :: \text{tail}(\Sigma(a_1)) \)  

\( \lambda' = A^\text{lam}_\Sigma(\lambda, j, \text{tail}(\Sigma(a_1)), \ldots, \text{tail}(\Sigma(a_n)))\) value

**Figure 20.** \( \text{filterT} \) without duplicating work.

\( A^\text{exp}\) (let \( \bar{v}^{(n)} = \text{reduceT}(\lambda, \{ e^{(n)} \}, \bar{a}^{(n)}) \) in \( e \)) =  

let \( \forall(v_1, \Sigma(e_1)), \ldots, \forall(v_n, \Sigma(e_n)) = \)  

reduceT(\( A^\text{lam}_\Sigma(\lambda, \bar{e}^{(n)}, \bar{p}^{(n)})\text{value}, \bar{e}^{(n)}, \bar{a}^{(n)} \)) in  

\( A^\text{exp}\) (let \( \bar{v}^{(n)} = \text{reduceT}(\lambda, \{ e^{(n)} \}, \bar{a}^{(n)}) \) in \( e \)) =  

\( \forall(v_1, \Sigma(e_1)), \ldots, \forall(v_n, \Sigma(e_n)) = \)  

reduceT(\( A^\text{lam}_\Sigma(\lambda, \bar{e}^{(n)}, \bar{p}^{(n)})\text{value}, \bar{e}^{(n)}, \bar{a}^{(n)} \)) in  

where \( r_i = \Sigma(e_i) \)  

\( p_i = \begin{cases} \Sigma(e_i) & \text{if } i \leq n \\ \text{tail}(\Sigma(a_i)) & \text{if } i > n \end{cases} \)  

\( \lambda_{\text{value}} = A^\text{lam}_\Sigma(\lambda, j, \text{tail}(\Sigma(a_1)), \ldots, \text{tail}(\Sigma(a_n)))\) value

**Figure 21.** The \( \text{reduceT} \) transformation rule (\( \text{scanT} \) is similar).

The 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, 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 \( f_{call} \). 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 \text{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.
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 GPGPU. Initial examples of such libraries include Nikola [27], a Haskell library for targeting Nvidia CUDA GPU. 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
References


