HERO-ML Specification
Version 0.126 *

Björn Lisper, Linus Källberg

February 7, 2023

Abstract

HERO-ML is a data-parallel array language, on very high level, which is intended to specify data parallel algorithms in a concise and platform-independent way. The goal is to support the software development for heterogeneous systems with different kinds of parallel numerical accelerators, where programs tend to be very platform-pecific and difficult to develop. Here we give a specification of HERO-ML including an abstract syntax, and an operational semantics. The part of the operational semantics that deals with arrays is in the form of sequential pseudocode, and can be seen as a specification of a reference implementation in a high-level intermediate format. Such a reference implementation exists, and we give a brief description of it.

*This research was funded by the KK-foundation through the HERO project, under grant no. 20180039.

Contents

1 Introduction

There is an ever-growing need for computational power. An area where the need for heavy computing is increasing rapidly is embedded systems, where applications like autonomous vehicles require massive amounts of computing for tasks like machine learning, advanced signal and image processing, etc. These systems are often real-time systems, and they sometimes have strong constraints on energy consumption, memory, unit cost, etc. The response from the hardware industry has been to develop increasingly integrated, heterogenous hardware, where *computational accelerators* are placed on the chip or board to offload computationally heavy tasks from the main processor. In this way, large computational resources can be provided at low cost.

Today we see a large proliferation of accelerator architectures: GPGPU's (like NVIDIA TESLA[1]), many-cores (such as Adapteva Ephiphany[2]), solutions involving FPGA (e.g., Xilinx ZYNQ[3]), and even ASICs. Although these architectures are quite different, they have in common that they typically rely on massive data parallelism to obtain performance. Besides embedded systems, these kinds of accelerators are also increasingly being used in traditional HPC as well as cloud computing: an example of the latter is Microsoft's Catapult project that integrates FPGAs into servers[4].

Developing software for these heterogenenous systems provides a challenge. Utilizing the accelerators well requires parallel code, but parallel programming can be very hard and error-prone. The situation is aggravated by the fact that current programming practices for the accelerators are very dependent on the type of accelerator. For instance, code for a GPGPU will typically be very different from code for a many-core. This makes the code less portable, and costly redesigns may be needed if the hardware platform is changed.

A possible way forward is to consider *model-based development*, where system and software is specified by high-level models rather than explicit program code. The models can capture different aspects such as system structure, or program logic. If the model is *executable* then it can be used to simulate the aspect of the system that it captures: such models can be used to capture errors in the design early, and they can also be used as test oracles in the validation phase.

HERO-ML is a data-parallel executable modeling language, intended to be used for very high level specifications of data parallel algorithms. Such executable specifications can serve as portable "blueprints" when developing accelerator code, and they can help finding flaws in the algorithms at an early stage of development. HERO-ML is inspired by data-parallel and array languages such as *lisp [10], NESL [1], ZPL [2], and HPF [7]. These languages all implement a parallel model of computation where the parallelism resides in collective operations over data structures, such as arrays, rather than explicitly in threads or processes. This model of computation is conceptually much simpler than the control parallelism given by threads and processes, and languages like the ones mentioned above introduce various high-level concepts and constructs that help writing clear and concise data parallel code. HERO-ML aims to generalize and unify these concepts. Since HERO-ML is intended for high-level modeling rather than high performance production code, its design does not have to make compromises in order to allow for efficient implementations. Thus, its design can rather focus on providing maximal support for the early modeling phase in the design of software for parallel accelerators.

2 HERO-ML Overview

HERO-ML is an imperative language extended with a data parallel array data type. The language is deliberately kept simple, since its main purpose is to demonstrate the principles of very high-level data parallel programming rather than providing a full-fledged production language. The sequential part is a standard WHILE language [9], extended with a type for so-called *abstract*

[1] `https://www.nvidia.com/en-us/data-center/tesla/`
[2] `https://www.adapteva.com/introduction/`
[3] `https://www.xilinx.com/products/silicon-devices/soc.html`
[4] `https://www.microsoft.com/en-us/research/project/project-catapult/`

arrays (see Section 2.1), a set of *array expressions*, a statement to *evaluate* such expressions, and bind the resulting array to a program variable, and a *masked concurrent assignment* where all elements in an existing array that fulfil some condition are updated.

As mentioned, HERO-ML is deliberately kept simple. Thus the current version lacks some features found in full programming languages. Features that are left out include user-defined functions and procedures, user-defined types, records, objects, and pointers. It is possible that some of these features will be added in future versions of HERO-ML.

2.1 Abstract Arrays, and Bounds

Abstract arrays are basically the same as the previously considered *data fields* [4, 5]. They provide a generalization of conventional arrays by observing that these really are partial functions from some domain of indices to a range of values. For conventional, dense arrays the domains are intervals. Abstract arrays generalise this by also allowing other domains. For instance we can have *sparse* arrays, whose domains are general finite sets. In general an abstract array is a pair

$$(f, bnd)$$

where *bnd* is its *bound*, which is a set representation, and f is a function defining the values of the array elements for the indices within the bound. Evaluating an abstract array means to first compute its bound, and then create a table with the array elements that represent the function. For this to work, the bound must represent a finite set. Some Hero-ML bounds indeed represent infinite sets, and array expressions with such bounds cannot be evaluated. (Infinite bounds may seem useless, but the evaluation of abstract arrays has a lazy flavor where such bounds can make sense.)

HERO-ML supports the following kinds of bounds:

- *dense bounds*, representing intervals,
- *sparse bounds*, representing general finite sets,
- *predicate bounds*, representing (possibly infinite) sets defined by a predicate,
- *empty* and *all*, representing the empty and universal set, respectively, and
- *product bounds* representing cartesian products.

Some two-dimensional bounds are illustrated in Fig. 1. In Section 9.1.2 we introduce a generalization of sparse multi-dimensional bounds, where the bound is embedded into a higher-dimensional space where it constrains certain dimensions. These embedded sparse bounds enables a considerably more precise handling of bounds for multi-dimensional sparse arrays.

The HERO-ML bounds form a *complete lattice* [9], which means that they have certain mathematical properties. Indeed there is a strong relation to static program analysis, and the computation of bounds can be seen as a kind of run-time value analysis. Some two-dimensional bounds are illustrated in Fig. 1.

There are three major kinds of array expressions in HERO-ML:

- *explicit array expressions*, which define arrays through listing their elements,
- *array comprehensions*, which define arrays with explicit bounds where the elements are computed according to some rule, and
- *forall expressions*, which are similar to array comprehensions but have their bounds defined implicitly from the syntax of the expression. This construct is inspired by lambda abstraction in the lambda calculus, and the rules for computing bounds are designed to (over)approximate the domain of the partial function defined by the forall exoression. Forall expressions, and the rules for computing their bounds, are further described in Section 9. (A variation of $forall$ expressions was also present in Data Field Haskell. See [4] for details.)

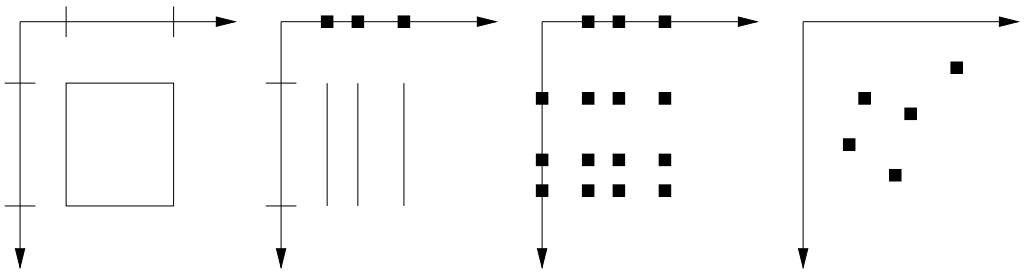

Figure 1: A dense product bound, a product of a dense and a sparse bound, a product of two sparse bounds, and a 2D sparse bound. (Adapted from [5].)

In addition there is a masked concurrent assignment of arrays, where array elements that fulfil some condition are concurrently updated. Together, these array constructs can express basically all data parallel constructs found in the literature.

An array access out of bounds returns an undefined value, which we denote "?". However also array elements within bounds may hold this value. This is since bounds may be over-approximated: in particular, the rules for deriving implicit bounds for forall expressions may produce such bounds. This has some consequences for collective operations such as *reduce* and *scan*, which will have to skip such elements. "?" is not accessible at source code level, but HERO-ML has a predicate $isDef$ that tests whether or not its argument is different from "?" (see Section 4.4).

2.2 Types

HERO-ML is a *strongly typed* language (all parts of a program must be well-typed for the program to be well-typed). It is *explicitly typed* (all declared entities must be given a type). The types are similar to those found in ML-like functional languages, such as OCaml or Haskell, but with some restrictions. It has basic types int, $float$, $bool$, a polymorhpic type $Array\ \iota\ \alpha$ for abstract arrays with indices of type ι and elements of type α, and a type $Bounds\ \iota$ for bounds over indices of type ι. α can itself be an array data type: thus, HERO-ML supports nested arrays. Array indices are either integers (for one-dimensional arrays), or tuples of integers (for multi-dimensional arrays). There is, however, no full-fledged data type for tuples: they appear only as indices to arrays, within expressions defining bounds, or as arguments to builtin functions and operators, see below.

HERO-ML is not a full-fledged higher-order language. However, builtin functions and operators have function types. Contrary to higher-order functional languages these functions are on uncurried form, taking a tuple of individual arguments as argument. For instance the operator "+" has the type $(int, int) \rightarrow int$, indicating that it takes two arguments of type int and returns an int. Arithmetic operators are overloaded over the numerical types int, and $float$: thus, for example,"+" also has the type $(float, float) \rightarrow float$. Implicit type conversion is not permitted: thus, numerical arguments of different type are not allowed for these operators. HERO-ML also has collective array operations *reduce*, and *scan*, which are higher-order in that they take a binary function, or operator, as an argument.

3 HERO-ML Syntax

We now specify the format of HERO-ML by an abstract syntax. This syntax is on syntax-tree level, where the trees are formal expressions, rather than on string level. These trees are close to parse trees, but with details abstracted away.

A HERO-ML program consists of two parts: a set of *type declarations*, where each user-defined

identifier is given a type, and a *statement* that provides the code to be executed:

$$prog \rightarrow typedecls; s$$

Here $typedecls$ is a list of type declarations, and s is a statement as defined below. The syntax for type declarations is given below:

$$\begin{array}{rcl} typedecls & \rightarrow & \Lambda \mid typedecl; typedecls' \\ typedecl & \rightarrow & id : type \\ type & \rightarrow & int \mid float \mid bool \mid Array\ index\_type\ type' \\ index\_type & \rightarrow & int \mid (int, int) \mid (int, int, int) \cdots \end{array}$$

Here id is an identifier, and $type$ stands for the types that can be assigned to user-declared entities (program variables in the current version of HERO-ML). The ellipsis for index types means that we allow n-tuples of int for any $n > 1$ as array indices. Thus, HERO-ML has arrays of any (finite) dimensionality. Λ stands for the empty expression (similar to the empty string for grammars defining sets of strings).

The formats for identifiers, and numeric literals, are given in Section 3.1.

We now define the statements of HERO-ML. We start with the sequential part, which is fairly standard (except that array expressions $aexp$, and bounds expressions bnd, defined later, may appear within "scalar" expressions):

Integer constant m, n, **floating-point constant** c

Integer variable i, **floating-point variable** f, **boolean variable** b

Array variable a

Bounds variable d

"Scalar" variable $y \rightarrow i \mid f \mid b$

Arithmetic operator $aop \rightarrow + \mid - \mid * \mid /$

General n**-ary function** fun

Integer expression

$$\begin{array}{rcl} iexp & \rightarrow & n \mid i \mid iexp\ aop\ iexp \\ & \mid & fun(exp_1, \ldots, exp_n) \mid aexp\ index\_exp \end{array}$$

Floating-point expression

$$\begin{array}{rcl} fexp & \rightarrow & c \mid f \mid fexp\ aop\ fexp' \\ & \mid & fun(exp_1, \ldots, exp_n) \mid aexp\ index\_exp \end{array}$$

Boolean operator $bop \rightarrow || \mid \&\&$

Relational operator $rop \rightarrow = \mid <$

Boolean expression $bexp$:

$$\begin{array}{rcl} bexp & \rightarrow & true \mid false \mid b \\ & \mid & iexp\ rop\ iexp' \mid fexp\ rop\ fexp' \mid bexp\ bop\ bexp \\ & \mid & fun(exp_1, \ldots, exp_n) \mid aexp\ index\_exp \end{array}$$

"Scalar" expression $sexp \rightarrow iexp \mid fexp \mid bexp$

Expression $exp \rightarrow sexp \mid aexp \mid bnd$

HERO-ML statements s:

$$\begin{array}{rcl} s & \rightarrow & skip \mid y = sexp \\ & \mid & s; s' \mid if\ bexp\ then\ s\ else\ s' \mid while\ bexp\ do\ s \\ & \mid & d = bnd \\ & \mid & a\_assign \end{array}$$

We now turn to the array-specific part. We define abstract syntax for *bounds expressions*, *array expressions*, and *array assignments*. But first we define the syntax for *array arguments* and *set elements*, respectively:

Index expressions $index\_exp$:

$$index\_exp \rightarrow [iexp] \mid [iexp_1, \ldots, iexp_m]$$

Set elements, and -expressions set_el_var, set_el_exp:

$$\begin{array}{rcl} set\_el\_var & \rightarrow & i \mid (i_1, \ldots, i_m) \\ set\_el\_exp & \rightarrow & iexp \mid (iexp_1, \ldots, iexp_m) \end{array}$$

Bounds expression bnd:

$$\begin{array}{rcl} bnd & \rightarrow & empty \mid all \mid d \mid iexp..iexp' \mid \{set\_el\_exp_1, \ldots, set\_el\_exp_m\} \\ & \mid & \{set\_el\_var : bexp\} \mid (bnd_1, \ldots, bnd_n) \\ & \mid & fun(exp_1, \ldots, exp_n) \mid aexp\ index\_exp \end{array}$$

Array expression $aexp$:

$$\begin{array}{rcl} aexp & \rightarrow & a \mid [exp\ :\ set\_el\_var\ in\ bnd] \\ & \mid & forall\ set\_el\_var \rightarrow exp \mid aexp|bnd \mid expl\_array \\ & \mid & fun(exp_1, \ldots, exp_n) \mid aexp\ index\_exp \end{array}$$

Explicit array expression $expl\_array$:

$$\begin{array}{rcl} db & \rightarrow & iexp..iexp' \mid iexp.. \mid ..iexp \\ db' & \rightarrow & \Lambda \mid db \\ preamble & \rightarrow & \Lambda \mid db : \mid (db'_1, \ldots, db'_n) \\ elist(1) & \rightarrow & exp_1, \ldots, exp_k \\ elist(n) & \rightarrow & elist(n-1); \ldots; elist(n-1);\ n > 1 \\ expl\_array & \rightarrow & [preamble\ elist(n)] \\ & \mid & [set\_el\_exp_1 : exp_1, \ldots, set\_el\_exp_n : exp_n] \end{array}$$

Explicit arrays, as the name suggests, have their elements explicitly given. The syntax allows the specification of both dense and sparse arrays, also multi-dimensional. Dense arrays are specified through lists of elements, which are nested for higher-dimensional arrays. Sparse arrays are defined by a list of index/element pairs.

Array assignment a_assign:

$$\begin{array}{rcl} a\_assign & \rightarrow & a\ =\ aexp \\ & \mid & a\ index\_exp_1 \cdots index\_exp_n\ =\ exp \\ & \mid & foreach\ set\_el\_var\ in\ bnd\ do\ a\ index\_exp_1 \cdots index\_exp_n\ =\ exp \end{array}$$

We now informally describe each new construct for creating bounds, abstract arrays, and assignment of such arrays. We also provide examples of their use.

Bounds:

- *empty*, *all*

  Description: corresponds to the empty and universal set, respectively.
  Example: `empty`, `all`
  Explanation:

- *d*

  Description: variable holding an already evaluated bound.
  Example:
  Explanation:

- *iexp..iexp′*

  Description: an interval bound (also called *dense bound*).
  Example: `1..10`
  Explanation: denotes the interval from `1` to `10`.

- $\{set\_el_1, \ldots, set\_el_m\}$

  Description: a finite, possibly irregular set, called a *sparse bound*.
  Example: `{(0,1),(3,2),(0,-1),(2,2)}`
  Explanation: a sparse two-dimensional bound with four elements.

- $\{set\_el\_var : bexp\}$

  Description: a so-called *predicate bound*, a possibly infinite set defined by a predicate.
  Example: `{(i,j) : i+j > 0}`
  Explanation: the set of all `(i,j)` such that `i+j > 0`.

- $(bnd_1, \ldots, bnd_n)$

  Description: cartesian product bound formed from n bounds.
  Example: `(1..10,1..25)`
  Explanation: the 2-D bound for a 10×25 dense matrix, formed from two 1-D bounds.

- *aexp index_exp*

  Description: access of an element in an array of bounds
  Example: `[i..i+5 : i in 1..10][3]` = `3..8`
  Explanation:

In addition there are a number of functions that return bounds. See Section 4.

Abstract Arrays:

- *a*

  Description: array variable holding an already evaluated array.
  Example:
  Explanation:

- [*exp* : *set_el_var in bnd*]

Description: array comprehension (definition of array with explicit bound).

Example: `[2*i : i in 1..10]`

Explanation: array with bound `1..10`, and elements `2*i`.

- *forall set_el_var* $\rightarrow$ *exp*

  Description: array definition with implicit bounds, and syntax similar to lambda-abstraction. (Rules for how to compute the bounds are given in Section 9.)

  Example: `forall i -> a[i] + b[i]`

  Explanation: elementwise addition of `a` and `b`, computing the bound from those of `a` and `b`.

- *aexp* | *bnd*

  Description: subarray of *aexp* defined by *bnd*.

  Example: `a | 1..10`

  Explanation: the subarray of `a` from `1` to `10`.

- *aexp index_exp*

  Description: access of an element in a possibly nested array of arrays.

  Example: `a[3]`

  Explanation: selection of element three, which is an array, from an array of arrays.

- [*preamble elist*(n)]

  Description: dense, possibly multidimensional, explicit array expression. "*preamble*" specifies the (possibly multi-dimensional) dense bounds. Bounds can be omitted, and then defaults to a bound with lower limit 0. "*elist*(n)" specifies an n-dimensional array, and it is a possibly nested, semicolon-separated list of lists where the lowest level specifies the array elements in comma-separated lists.

  Example 1: `[2.. : 1,3,2]`

  Explanation: a dense one-dimensional array with bound `2..4`, and three elements `1`, `3`, `2`. Elements are explicitly given, hence the name.

  Example 2: `[..4 : 1,3,2]`

  Explanation: same array as above, but specified through the upper limit for the interval.

  Example 3: `[1,3,2]`

  Explanation: an array with the same elements as above, but bound `0..2` (similar to an array in C).

  Example 4: `[(1..2,1..3) : 1,2,3; 4,5,6;]`

  Explanation: a 2D-array (matrix) with rows `1,2,3`, and `4,5,6`. (The last semi-colon can be dropped.)

  Example 5: `[1,2,3; 4,5,6;]`

  Explanation: a 2D-array like in Example 4, but with default bounds `(0..1,0..2)`.

  Example 6: `[(,,98..100) : 1,2,3; 4,5,6;; 7,8,9; 10,11,12;; 13,14,15; 16,17,18;;]`

  Explanation: a 3D-array with bounds `(0..2,0..1,98..100)`. (The two first are default bounds.) If we name the array "`A`", then `A[1,0,99] = 8`.

- [*set_el_exp*$_1$: *exp*$_1$, ..., *set_el_exp*$_n$: *exp*$_n$]

  Description: explicit sparse array expression.

Example: `[(1,1):4.7, (2,3):0.01, (3,5):3.14]`

Explanation: a sparse two-dimensional array with bound {`(1,1), (2,3), 3,5)`}, and three elements `4.7`, `0.01`, `3.14`. Like for dense explicit arrays bound and elements are explicitly given, hence the name. It is an error to define the value for the same index more than once.

In addition there are a number of functions that return arrays. See Section 4.

Abstract Array Assignments:

- a = *aexp*

  Description: creates a new abstract array by evaluating *aexp*, and sets a to hold it.

  Example: `a = [2*i : i in 1..10]`

  Explanation: sets `a` to the new array defined by the array expression.

- a *index_exp*$_1 \cdots$ *index_exp*$_n$ = *exp*

  Description: Assignment of an array element (including assigning an array within a nested array).

  Example: `a[j] = [2*i : i in 1..10]`

  Explanation: sets `a[j]` to the new array defined by the array expression. (Here, `a` is an array of arrays.)

- *foreach set_el_var in bnd do* a *index_exp*$_1 \cdots$ *index_exp*$_n$ = *exp*

  Description: destructive, masked update where a *index_exp*$_1 \cdots$ *index_exp*$_n$ is set to *exp* for all values of *set_el_var* that belong to *bnd* and where *exp* is defined. Note that if $n > 1$ or $m > 1$ then the corresponding array is nested.

  Example (not nested): `foreach i in 1..10 do x[i+1] = y[i-1]`

  Explanation: set `x[i+1]` to `y[i-1]` for all `i` in `1..10` where `y[i-1]` is defined.

  Example (nested): `foreach i in 1..10 do x[i][i+1] = y[3][i-1]`

  Explanation: set `x[i][i+1]` to `y[3][i-1]` for all `i` in `1..10` where `y[3][i-1]` is defined.

  Note: it might be that the same element in the left-hand side is targeted by the right-hand side for more than one value of `i`. This implies a write conflict, and the value of the targeted array element is then non-deterministically set to one of the values written there. An error also occurs if, for some `i`, the target address for the left-hand side is out of bounds.

  Example (write conflict): `foreach i in 1..10 do x[1] = y[i]`

  Explanation: here there is an attempt to write several elements of `y` to `x[1]`.

  Example (write out of bounds): `foreach i in 1..10 do x[i] = y[i]`, where `bound(x)` = `1..9`

  Explanation: if `y[10]` is defined then there is an attempt to write it to `x[10]`, which is out of bounds.

3.1 HERO-ML Concrete Syntax

We now address the most relevant aspects of the concrete syntax of HERO-ML that either differ from or are left unspecified in the abstract syntax used throughout the rest of this document. To begin with, the part of HERO-ML dealing with scalar computations mostly adheres to the "standard" syntax rules found in most conventional languages such as C. Specifically this means that infix notation is used for the arithmetic, relational, and logical operators `+`, `-`, `*`, `/`, `%` (integer remainder), `<`, `<=` and so on, and that these follow the usual operator precedence and associativity rules. Also, integer, floating-point, and boolean literals as well as variable names all follow standard formats, with the exception that variable names are allowed to include a trailing sequence of single quotes, as in, e.g., `x'` and `y''`, which is somewhat less conventional.

The language only defines two specific infix operators for bounds and arrays, and these are the "`|`" operator used to perform "array slicing", and the double-dot operator "`..`" used to specify dense bounds. Insofar as these operators need precedences and associativities to resolve syntactical ambiguities, these have mostly been chosen in the natural way given the type rules of the language. For example, for an ambiguous expression like $arr \,|\, bnd_1 \,|\, bnd_2$, only the interpretation $(arr \,|\, bnd_1) \,|\, bnd_2$ makes sense, since the alternative interpretation $arr \,|\, (bnd_1 \,|\, bnd_2)$ would give rise to a type error in the second slicing operation. However, one true source of ambiguity is when a $forall$ expression is immediately followed by an array slicing operation. For example, the expression $forall\ i \to arr[i] \,|\, bnd$ could be interpreted either as $(forall\ i \to arr[i]) \,|\, bnd$ or as $forall\ i \to (arr[i] \,|\, bnd)$, where the second variant can make sense if arr is an array of arrays. This ambiguity has been resolved in favor of the first variant, as using the `|` operator to slice the result of a $forall$ expression should be the more common use case. In the case of nested forall expressions followed by a slicing operation, $forall\ i \to forall\ j \to \ldots \,|\, bnd$, an ambiguity arises as to which $forall$ expression the slicing should be attached to. Analogously to how the classical "dangling else" problem is commonly resolved, HERO-ML lets this "dangling slice" be attached to the rightmost forall expression.

Similarly to some languages such as Python and F#, HERO-ML is sensitive to how the code is indented to allow for a more lightweight syntax, where statements and variable declarations can be terminated with simple line breaks as opposed to explicit characters like semicolons, and the block structure of the program can be expressed by changes in indentation instead of parentheses or special keywords. At the beginning of each new block—that is, after the "do" keyword in a while statement or after the "then" and "else" keywords in an if-else statement—the first statement to appear sets a base indentation for the rest of the block. This first statement, which is allowed to be on the same line as the preceding keyword, must be at least one character position to the right of the indentation of the surrounding block, otherwise the block is considered empty. Then the indentation of each subsequent line of code (ignoring blank lines) is compared against the block's base indentation, and the following version of the "offside rule" is applied:

- If a line starts at a column number that coincides with the current indentation, then it is considered the beginning of a new statement inside the current code block. This is as if an invisible semicolon character had appeared to mark the end of the statement on the previous line. It is a syntax error, of course, if the statement on the previous line was not syntactically complete.

- If the new line starts strictly to the right of the indentation, then it is instead counted as a continuation of the previous line, as if no line break had appeared at all. This way long statements can be split over multiple lines by having the continuation lines be indented relative to the first line.

- If a line starts strictly to the left of the current indentation, then this marks the termination of the current code block (and of the previous statement). The same rules are then applied again to the same line but in the surrounding block and relative to that block's base indentation. Note that this can lead to the closing of multiple nested blocks at the same time.

Mainly the same principles also apply to the global scope of the program, where the first non-blank line in the program sets the indentation of the global scope. However, an exception to the last rule above applies if the base indentation is non-zero, as there is no surrounding block to exit to in this case. Then the base indentation is instead simply adjusted to be the same as that of the "offside" line from that point on.

An exception is also made for expressions enclosed in parentheses (including square brackets and curly braces). As there are no statements or declarations inside of the parentheses to delimit, the indentation rules are simply suspended there, up to and including the closing parenthesis, meaning that all indentation and line breaks are ignored altogether. The main motivation for this is to allow a more liberal use of indentation and line breaking when formatting explicit matrix expressions in a program. Finally, the syntax allows for semicolons as an alternative way to delimit statements, which makes it possible to put multiple statements on the same line.

4 HERO-ML Functions

The HERO-ML grammar has a case $fun(exp_1, \ldots, exp_n)$ for functions. This case is matched by a number of functions in HERO-ML, where some provide important features. We therefore give an account here for the functions in HERO-ML.

4.1 Numerical Functions

HERO-ML will come with a set of numerical functions, where the argument and result types are numerical. The exact selection of functions is not decided yet, but will be very much standard including standard scientific functions.

4.2 Boolean Functions, and Operators

$not : bool \to bool$ provides logical negation.

The boolean operators ||, && have a semantics where their arguments are evaluated from left to right, and the evaluation is stopped as soon as the outcome is known. Thus, they are *non-strict* in their second argument since it is not always evaluated. (Non-strict here means that the function may return a defined value even if the argument is undefined.)

4.3 Conditional Function

HERO-ML has a conditional function $if : (bool, \alpha, \alpha) \to \alpha$, where α is a type variable.

Example: `if(x > 0,5,7)` evaluates to `5` if `x > 0` in the current environment, and to `7` otherwise.

4.4 Test for Undefined Value

$isDef : \alpha \to bool$ tests whether or not its argument is different from the undefined value "?".

Examples: `isDef(17)` = `true`, `isDef(?)` = `false`.

4.5 Functions on Abstract Arrays

$bound : (Array\ \iota\ \alpha) \to Bound\ \iota$. $bound(a)$ extracts the bound from the array a.

Example: `bound([2..4 : 1,3,2])` = `2..4`.

$reduce : ((\alpha, \alpha) \to \alpha, Array\ \iota\ \alpha) \to \alpha$. $reduce(f, a)$ "sums" the elements in the array a using the binary function/operator f.

Example: `reduce(+,[2..4 : 1,3,2])` = `1 + 3 + 2 = 6`.

scan : $((\alpha, \alpha) \to \alpha, \mathit{Array}\ \iota\ \alpha) \to \mathit{Array}\ \iota\ \alpha$. Like *reduce*, but computes an array with all the"partial sums". The bound is the same as for a, and the "partial sums" are computed in the lexicographic order over $\mathit{bound}(a)$.

Example: `scan(+,[2..4 : 1,3,2]) = [2..4 : 1,4,6]`

4.6 Functions on Bounds

HERO-ML has many functions on bounds. Many of these are used to implement the implicit derivation of bounds that takes place when array-valued expressions, such as *forall*-expressions, are evaluated. They are exposed at the user level primarily to allow the manual tailoring of bounds in cases when the implicit bounds derivation does not yield a satisfactory result. The functions can also be seen as providing an *interface* for bounds: any set representation that implements these functions (such that they have certain prescribed properties) can be used as bounds.

member : $(\iota, \mathit{Bounds}\ \iota) \to \mathit{bool}$: $\mathit{member}(i, \mathit{bnd})$ returns *true* when i is a member of the set defined by *bnd*.

$\mathit{join}, \mathit{meet} : (\mathit{Bounds}\ \iota, \mathit{Bounds}\ \iota) \to \mathit{Bounds}\ \iota$. Lattice-theoretical $\sqcup$ and $\sqcap$, respectively, in the lattice of bounds over $\mathit{Bounds}\ \iota$ (approximating set union and -intersection). Their semantics are given in Section 8.

Example (dense 1-D bounds): `meet(1..10,5..30) = 5..10`, `join(1..10,20..30) = 1..30`.

$\mathit{isDense}, \mathit{isSparse}, \mathit{isPredicate}, \mathit{isProduct} : (\mathit{Bounds}\ \iota) \to \mathit{bool}$: they all test whether their arguments are of the corresponding type.

$\mathit{finite} : (\mathit{Bounds}\ \iota) \to \mathit{bool}$: $\mathit{finite}(\mathit{bnd})$ is true iff *bnd* is classified as finite (*i.e.*, if *bnd* is dense, sparse, *empty*, or a product of finite bounds).

$\mathit{size} : (\mathit{Bounds}\ \iota) \to \mathit{int}$: $\mathit{size}(\mathit{bnd})$ returns the number of elements in the finite bound *bnd*. It is undefined for infinite bounds.

$\mathit{enum} : (\mathit{Bounds}\ \iota) \to \mathit{Array}\ \mathit{int}\ \iota$: $\mathit{enum}(\mathit{bnd})$ provides an enumeration of the elements in the finite bound *bnd*, in lexicographic order, in the form of an array with bound $0..\mathit{size}(\mathit{bnd}) - 1$ containing the elements of *bnd* in this order. Undefined for infinite bounds.

Example (sparse 1-D bound): `enum({3,1,7}) = [0..2 : 1,3,7]`.

Note: *enum* is currently not visible in HERO-ML. It is however visible (and used) in intermediate layers. It is also used in the semantics for functions such as *reduce*, and *scan*.

5 I/O

To allow programs to communicate with the surrounding environment, HERO-ML provides a simple model where programs have access to an input pipe and an output pipe. HERO-ML values can be read using the "*in*" operator, and written using an *out* statement. In the abstract syntax in Section 3 the grammars for expressions, and statements are extended with the following cases:

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{in}\ \mathit{type} \\ s & \to & \mathit{out} \mid \mathit{out}\ \mathit{exp} \mid \mathit{out}\ \mathit{exp}_1, \ldots, \mathit{exp}_n \end{array}$$

Thus *in type* reads a HERO-ML value of type *type* from the input pipe, and *out exp* writes the value of *exp* to the output pipe. *out* can also write multiple expressions given as a comma-separated list, and it can also be executed without any arguments in which case a kind of marker is written to the output pipe.

in has to be typed since HERO-ML is a strongly and explicitly typed language. Reading a value of the wrong type will give a runtime error.

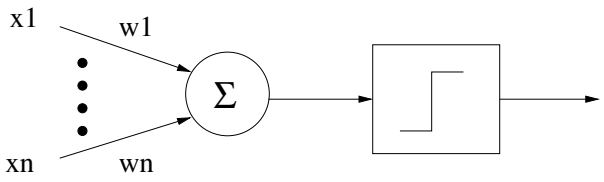

Figure 2: McCulloch-Pitts neuron.

Values of any type can be read and written: however they will always be evaluated. For example, the statement *out* $x + y$ evaluates the expression $x + y$ and then writes the result to the output pipe. An example of the use of *in* is the following:

$$x = 23 + in\ int * y$$

Here an integer value is read from the input pipe, it is multiplied by the variable y, 23 is added, and then result is stored in the variable x.

HERO-ML only knows the *in* and *out* primitives for communicating with the environment. It hs no means to connect the input and output pipes to any particular channels. What these pipes are connected to can therefore be adapted outside HERO-ML for the specific environment. For example, an interpreter/debugger for the language could simply let the data coming through the output pipe be written to the screen or to a log file, and it could similarly have the input pipe be connected to a user prompt or a file. On the other hand, when translating a HERO-ML program to code for some target platform, the pipes would typically be mapped to the specific I/O mechanisms of that platform. For instance, if a program were to be translated into a CUDA kernel, then the pipes could be mapped to input/output parameters of the kernel.

6 A Worked Example: Feed-Forward ANN

We now show how the computation performed by a classical feed-forward network can be modeled in HERO-ML.

6.1 Background

An Artificial Neural Network (ANN) is a computational structure whose way of working is inspired by how the nervous system works in living beings. Such nervous systems typically have the following characteristics:

- they are composed out of many *small*, *interconnected units* (the *neurons*),
- the interconnection is typically *sparse*, i.e., each neuron is connected only to a few other neurons,
- each neuron has an *activity level*, which is a affected by the activity levels of the neurons that it is connected to, and
- the activity level of a neuron is a *highly non-linear* function of the activity levels of its connected neurons. In particular there are *thresholding effects* caused by the activity level being saturated.

ANN's are mimicking this. There are many variations. A claassical example is the *McCulloch-Pitts neuron*, where the activity level of a neuron is computed by thresholding a weighted sum of the activity levels of its neighbours. See Fig. 2.

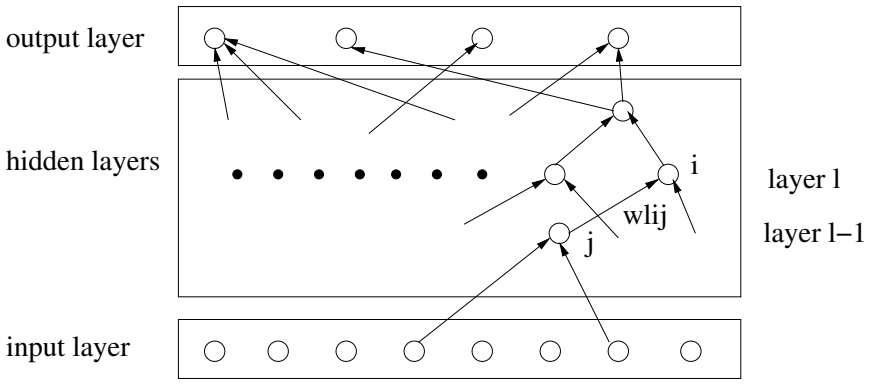

Figure 3: Layers in a feed-forward network.

6.2 Feed-Forward Networks

In a *feed-forward network* the units are arranged in a number of *layers*, where each interconnection goes from some layer $l-1$ to layer l. See Fig. 3. There are three kinds of layers:

- an *input* layer, which provides the input to the ANN,
- a number of *hidden* layers, which contain units interconnected between layers similarly to the McCulloch-Pitts neuron in Fig. 2, and
- an *output* layer, which provides the output (or response) from the ANN given a certain input.

The input is basically an array of numbers. It can, for instance, be a pixel matrix encoding a picture. The output is also an array of numbers that encodes the output. It can, for instance, represent a classification of some object in the picture.

Each unit in a hidden layer computes its output as a weighted, thresholded sum of the outputs of the connected units in the previous layers, similarly to the McCulloch-Pitts neuron in Fig. 2. Mathematically, output z_{lj} from unit j in layer l is computed as

$$z_{li} = s(\sum_j w_{lij} z_{l-1j}) \tag{1}$$

where the sum ranges over the units j in layer $l-1$ that are connected to unit i in layer l. w_{lij} is the *weight* of the connection from j in layer $l-1$ to i in layer l. s is commonly chosen as the *sigmoid function*, defined by

$$s(x) = \frac{1}{1+e^{-x}} \tag{2}$$

The sigmoid function provides a "soft" thresholding, see the sketch in Fig. 4.

The weights are very important. They provide the knowledge that is stored in the network. *Training* the network means to set the weights in order to have a response from the network that is as close as possible to the desired output. A feed-forward network with hidden layers is a kind of *Deep Neural Network* (DNN), and training such a network is called *deep learning*. There are systematic methods, such as *back-propagation*, to do this, but we will not treat this further here.

6.3 Modeling of Feed-Forward Network Computing with Hero-ML

We will now show how to model one particular way of computing the output from a trained feed-forward network, given some input. We will use nested arrays, where the nesting reflects the structuring of the network into layers. More specifically `z` will be an array of arrays, where `z[l]`

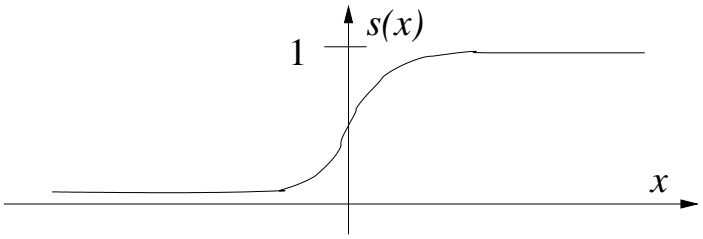

Figure 4: The sigmoid function.

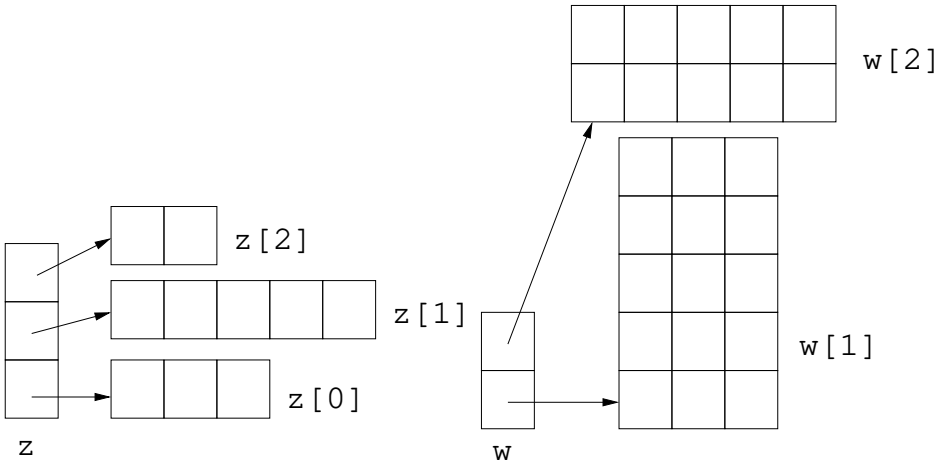

Figure 5: A nested array representation of layers in a feed-forward network.

holds the output values of the units in layer `l`. The weights will be stored in an array of matrices `w`, where `w[l]` is a matrix where each element `w[l][i,j]` holds the weight for the connection from unit `j` in layer `l-1` to unit `i` in layer `l`. The types of `z` and `w` are as follows:

```
z : Array int (Array int float)
w : Array int (Array (int,int) float)
```

Note that `w[l]` might be a sparse matrix, with a sparse bound. We assume that the arrays `z` and `w` themselves have dense bounds `0..n-1` and `1..n-1`, respectively, where `n` is the number of layers. Fig. 5 shows an example with an input layer with three units, a hidden layer with five units, and an output layer with two units. We can note that the nested array representation easily can handle the fact that different layers can hold different numbers of units in feed-forward networks.

We now give HERO-ML code for the computation. We use the following declarations[5]:

```
s(x) = 1/(1 + exp (-x)) // sigmoid function
sum(a) = reduce(+,a)    // sum over abstract array
```

We assume that the input to the computation is stored in the array `input`. First, the input layer `z[0]` is assigned this array. Then the code loops over the other layers, computing `z[l]` from a matrix-array multiplication of `z[l-1]` and `w[l]` followed by a thresholding of the elements in the resulting array:

```
z[0] = input;
l = 1;
```

[5]HERO-ML does not have function definitions, but we can see these declarations as macros.

```
  while l < n do
    z[l] = forall i -> s(sum(forall j -> (w[l][i,j] * z[l-1][j])));
    l = l + 1
```

This version creates a new abstract array for each array assignment. As an alternative we can instead use a `foreach` statement, which performs an in-place update:

```
foreach i in bound(z[0]) do z[0][i] = input[i];
l = 1;
  while l < n do
    foreach i in bound(z[l]) do
      z[l][i] = s(sum(forall j -> (w[l][i,j] * z[l-1][j])));
    l = l + 1
```

What if we instead choose to use flat (non-nested) arrays? `z` then turns ito a matrix, and `w` becomes a three-dimensional tensor. Their types will now be as follows:

```
z : Array (int,int) float
w : Array (int,int,int) float
```

For each iteration `l`, row `l` in `z` will now be updated. This is accomplished by a `foreach` statement where the elements to be updated are selected from this row. We obtain the following code:

```
foreach i in bound(forall j -> z[0,j]) do z[0,i] = input[i];
l = 1;
  while l < n do
    foreach i in bound(forall j -> z[l,j]) do
      z[l,i] = s(sum(forall j -> (w[l,i,j] * z[l-1,j])));
    l = l + 1
```

Here the `l`'th row is extracted using a `forall` expression.

A disadvantage with flat arrays is that if `z` has a dense bound (a regular matrix) then all layers will be modeled to have the same number of units. There are of course ways around this, but the nested array approach still seems to provide a better fit.

7 HERO-ML Semantics

We now present a sequential operational semantics for HERO-ML. The semantics is structured into two parts:

1. For statements, the semantics is given as rules for state transitions, and
2. for expressions we define an *eval* function that evaluates expressions relative to the current state.

The semantics for statements, excluding assignments involving abstract arrays, is quite standard. The rules for state transitions have the form $(s, S) \rightarrow S'$ where s is a statement, S is a function from program variables to values (a so-called *store*), expressing the current contents of the memory before executing the statement, and S' is a store expressing the memory contents after having executed s. We give this part in Fig. 6. (Masked update of abstract arrays will be treated later.)

Computations can return the undefined value "?". The semantics has special rules covering the cases where boolean conditions that affect the program flow become undefined. In such a case the program goes into a particular error state, where it halts.

Array indices and set elements can be either integers, or integer tuples. For simplicity we assume that they are integers in this section. The extension to integer tuples is straightforward, but tedious.

$$(skip, S) \to S \quad (y = sexp, S) \to S[y \mapsto eval(sexp, S, \emptyset)]$$

$$\frac{(s, S) \to S'' \quad (s', S'') \to S'}{(s; s', S) \to S'}$$

$$\frac{eval(bexp, S, \emptyset) = true \quad (s, S) \to S'}{(if\ bexp\ then\ s\ else\ s', S) \to S'} \qquad \frac{eval(bexp, S, \emptyset) = false \quad (s', S) \to S'}{(if\ bexp\ then\ s\ else\ s', S) \to S'}$$

$$\frac{eval(bexp, S, \emptyset) = true \quad (s, S) \to S'' \quad (while\ bexp\ do\ s, S'') \to S'}{(while\ bexp\ do\ s, S) \to S'} \qquad \frac{eval(bexp, S, \emptyset) = false}{(while\ bexp\ do\ s, S) \to S}$$

$$\frac{eval(bexp, S, \emptyset) = ?}{(if\ bexp\ then\ s\ else\ s', S) \to ERROR} \qquad \frac{eval(bexp, S, \emptyset) = ?}{(while\ bexp\ do\ s, S) \to ERROR}$$

$$(a = aexp, S) \to S[a \mapsto eval(aexp, S, \emptyset)]$$

$$\frac{i_j = eval(iexp_j, S, \emptyset) \in bound(a\ iexp_1 \cdots iexp_{j-1}), j = 1, \ldots, n}{(a\ iexp_1 \cdots iexp_n = exp, S) \to S[a[i_1] \cdots [i_n] \mapsto eval(exp, S, \emptyset)]}$$

Figure 6: Operational semantics for the sequential statements of HERO-ML.

The notation $S[y \mapsto v]$ is used to denote a store with the same contents as S except that $S[y \mapsto v](y) = v$. In the last transition rule in Fig. 6 we extend this notation in order to express update of array elements. This rule gives semantics to assignments of elements in possibly nested arrays. It simply expresses that first all array indices are evaluated, and if they are all within bounds then the corresponding array element is replaced with the value obtained by evaluating the right-hand side.

HERO-ML expressions are evaluated using the *eval* function. Usually this function takes two arguments: the expression to be evaluated, and a store that gives the current memory contents. Here *eval* also takes a third argument, which is a set of variables that are bound on some outer level. This turns out to be needed since HERO-ML has three variable-binding constructs: predicate bounds, array comprehensions, and `forall` expressions. Within such expressions *eval* might return values that are symbolic expressions in the bound variables. However note that on the top level, for statements, *eval* will always return a constant value.

We now define *eval* for the various kinds of expressions found in HERO-ML. We do this in a rewriting style, where each equation can be seen as a rewrite rule where (sub)expressions matching the left-hand side of the rule are transformed into the corresponding instance of the right-hand side. An expression that cannot be rewritten further is called a *normal form* (or *nf*). Simple constants are normal forms, but more complex expressions possibly containing both bound and free variables can also be normal forms. A normal form that contains no free variables is *closed*.

For the "standard" expressions of HERO-ML we skip the division of expressions into different types from Section 3. We use c for constants, x for variables, e for expressions, and f for n-ary functions except those that are otherwise defined. Every n-ary function f has its semantics defined by a number of equations $f(c_1, \ldots, c_n) = c$, where $c_1, \ldots, c_n$ are constants and c is some constant.

Let S be a store, and BV a set of bound variables. Then:

$$
\begin{aligned}
eval(c, S, BV) &= c \\
eval(x, S, BV) &= S(x),\ x \notin BV \\
eval(x, S, BV) &= x,\ x \in BV \\
eval(if(bexpr, e, e'), S, BV) &= eval(e, S, BV),\ eval(bexpr, S, BV) = true \\
eval(if(bexpr, e, e'), S, BV) &= eval(e', S, BV),\ eval(bexpr, S, BV) = false \\
eval(if(bexpr, e, e'), S, BV) &= ?,\ eval(bexpr, S, BV) = ? \\
eval(if(bexpr, e, e'), S, BV) &= if(eval(bexpr, S, BV), eval(e, S, BV), eval(e', S, BV)),\ otherwise \\
eval(bexpr \mid\mid bexpr'), S, BV) &= true\ if\ eval(bexpr, S, BV) = true \\
eval(bexpr \mid\mid bexpr'), S, BV) &= eval(bexpr', S, BV)\ if\ eval(bexpr, S, BV) = false \\
eval(bexpr \mid\mid bexpr'), S, BV) &= ?\ if\ eval(bexpr, S, BV) = ? \\
eval(bexpr \mid\mid bexpr'), S, BV) &= eval(bexpr, S, BV) \mid\mid eval(bexpr', S, BV),\ otherwise \\
eval(bexpr \,\&\&\, bexpr'), S, BV) &= false\ if\ eval(bexpr, S, BV) = false \\
eval(bexpr \,\&\&\, bexpr'), S, BV) &= eval(bexpr', S, BV)\ if\ eval(bexpr, S, BV) = true \\
eval(bexpr \,\&\&\, bexpr'), S, BV) &= ?\ if\ eval(bexpr, S, BV) = ? \\
eval(bexpr \,\&\&\, bexpr'), S, BV) &= eval(bexpr, S, BV) \,\&\&\, eval(bexpr', S, BV),\ otherwise \\
eval(f(e_1, \ldots, e_n), S, BV) &= c\ when\ eval(e_i, S, BV) = c_i\ for\ i = 1, \ldots, n \\
eval(f(e_1, \ldots, e_n), S, BV) &= f(eval(e_1, S, BV), \ldots, eval(e_n, S, BV))\ otherwise
\end{aligned}
$$

We assume that the functions f are "*?-strict*" in all their arguments, that is: ? is returned as soon as some argument is ?. Next, we define *eval* for bounds expressions:

$$
\begin{aligned}
eval(c, S, BV) &= c,\ c = empty, all \\
eval(d, S, BV) &= S(d) \\
eval(iexp..iexp', S, BV) &= eval(iexp, S, BV)..eval(iexp', S, BV) \\
eval(\{iexp_1, \ldots, iexp_m\}, S, BV) &= \{eval(iexp_1, , S, BV) \ldots, eval(iexp_m, S, BV)\} \\
eval(\{i : bexp\}, S, BV) &= \{i : eval(bexp, S, BV \cup i)\} \\
eval((bnd_1, \ldots, bnd_n), S, BV) &= (eval(bnd_1, S, BV), \ldots, eval(bnd_n, S, BV))
\end{aligned}
$$

We now define *eval* for array expressions *aexp*, and access of array element *aexp iexp*. The evaluation of such expressions is done in two steps. First, a function *eval′* is applied to *aexp*. This function is similar to *eval* for non-array expressions, replacing program variables with their current values and simplifying the resulting expressions. If the result is a closed normal form then the evaluation proceeds as follows: for array expressions that are to be fully evaluated, a function *tabulate* allocates memory and evaluates all array elements. Applied array expressions *aexp iexp*, on the other hand, are evaluated in a lazy fashion by the *eval_elem* function, which evaluates and returns exactly the indexed array element.

As mentioned, arrays are semantically pairs (f, b) where f is a function defining the array elements and b is a bound. We define $fun(f, b) = f$, and $bound(f, b) = b$. Below we extend these functions to general array expressions by defining instances of *eval* for them. These instances are used in the definitions of *tabulate*, and *eval_elem*.

The evaluated arrays produced by *tabulate* are pairs (t, b), where t is a table defining the function. We define $tab(t, b) = t$ in this case. Table elements are accessed with array notation, viz. $t[i]$ for element i. Array variables a always hold fully evaluated arrays: thus, $S(a) = (t, b)$ for some table t, and bound b. Fully evaluated arrays a are ?-strict, that is: $a[?] = ?$.

In addition to the previously introduced functions we assume that the following functions are available:

- $alloc(bnd, \tau)$: allocates memory for the table of an array with finite bound *bnd*, and elements of type τ all initialized to the undefined value "?". (We also assume a function $target\_type(aexp)$ that returns the type of the elements in the array defined by *aexp*.)
- $B(exp, i, Y)$: derives the bound for $forall\ i \to exp$ in a context where Y is the current set of variables that are bound in some enclosing variable-binding construct (like an outer $forall$). $B(exp, i, Y)$ is defined in Section 9.

We now define $eval'$:

$$\begin{array}{rcl}
eval'(a, S, BV) & = & S(a) \\
eval'([exp\ :\ i\ in\ bnd], S, BV) & = & [eval(exp, S, BV \cup \{i\})\ :\ i\ in\ eval(bnd, S, BV)] \\
eval'(forall\ i \to exp, S, BV) & = & forall\ i \to eval(exp, S, BV \cup \{i\}) \\
eval'(aexp \,|\, bnd, S, BV) & = & (eval(fun(aexp), S, BV), eval(bound(aexp) \sqcap bnd, S, BV) \\
eval'([iexp.. : exp_1, \ldots, exp_n], S, BV) & = & [eval(iexp, S, BV).. : eval(exp_1, S, BV), \ldots, eval(exp_n, S, BV)] \\
eval'([..iexp : exp_1, \ldots, exp_n], S, BV) & = & [..eval(iexp, S, BV) : eval(exp_1, S, BV), \ldots, eval(exp_n, S, BV)] \\
eval'([exp_1, \ldots, exp_n], S, BV) & = & [eval(exp_1, S, BV), \ldots, eval(exp_n, S, BV)] \\
eval'([iexp_1 : exp_1, \ldots, iexp_n : exp_n], S, BV) & = & [eval(iexp_1, S, BV) : eval(exp_1, S, BV), \ldots, eval(iexp_n, S, BV) : eval
\end{array}$$

Next we define $eval$ for $bound(aexpr)$:

$$\begin{array}{rcl}
eval(bound(a), S, BV) & = & bound(S(a)) \\
eval(bound([exp\ :\ i\ in\ bnd]), S, BV) & = & eval(bnd, S, BV) \\
eval(bound(forall\ i \to exp), S, BV) & = & B(eval(exp, S, BV \cup i), i, \emptyset) \\
eval(bound(aexp \,|\, bnd), S, BV) & = & eval(bound(aexpr) \sqcap bnd, S, BV) \\
eval(bound([iexp.. : exp_1, \ldots, exp_n]), S, BV) & = & let\ i = eval(iexp, S, BV)\ in\ i..(i + n - 1) \\
eval(bound([..iexp : exp_1, \ldots, exp_n]), S, BV) & = & let\ i = eval(iexp, S, BV)\ in\ (i - n + 1)..i \\
eval(bound([exp_1, \ldots, exp_n]), S, BV) & = & 0..(n - 1) \\
eval(bound([iexp_1 : exp_1, \ldots, iexp_n : exp_n]), S, BV) & = & \{eval(iexp_1, S, BV), \ldots, eval(iexp_n, S, BV)\}
\end{array}$$

Then we define $eval$ for $fun(aexpr)$:

$$\begin{array}{rcl}
eval(fun(a), S, BV) & = & \lambda i.(tab(a)[i]) \\
eval(fun([exp\ :\ i\ in\ bnd]), S, BV) & = & \lambda i.eval(exp, S, BV \cup \{i\}) \\
eval(fun(forall\ i \to exp), S, BV) & = & \lambda i.eval(exp, S, BV \cup \{i\}) \\
eval(fun(aexp \,|\, bnd), S, BV) & = & eval(fun(aexp), S, BV) \\
eval(fun([iexp.. : exp_1, \ldots, exp_n]), S, BV) & = & let\ i = eval(iexp, S, BV)\ in \\
 & & \quad \{\, k \mapsto eval(exp_k, S, BV) \mid i \le k \le i + n - 1 \,\} \\
eval(fun([..iexp : exp_1, \ldots, exp_n]), S, BV) & = & let\ i = eval(iexp, S, BV)\ in \\
 & & \quad \{\, k \mapsto eval(exp_k, S, BV) \mid i - n + 1 \le k \le i \,\} \\
eval(fun([exp_1, \ldots, exp_n]), S, BV) & = & \{\, k \mapsto eval(exp_k, S, BV) \mid 0 \le k \le n - 1 \,\} \\
eval(fun([iexp_1 : exp_1, \ldots, iexp_n : exp_n]), S, BV) & = & \{\, eval(iexp_k, S, BV) \mapsto eval(exp_k, S, BV) \mid 1 \le k \le n \,\}
\end{array}$$

We are now in a position where we can define *eval_elem*, and *tabulate*. The semantics for this part is defined using a different, more informal style, where the evaluation is expressed by a kind of pseudocode. This pseudocode is expressed in what could be a high-level intermediate format. In *tabulate* we use the notation $set(bnd)$ for the set of indices defined by the bound *bnd*:

$eval\_elem(aexp\ iexp, S, BV) =$
  $b = eval(bound(aexp), S, BV);$
  $f = eval(fun(aexp), S, BV);$
  $i = eval(iexp, S, BV);$
  $if\ not\ member(i, b)\ then\ ERROR/return\ ?$[6];
  $return\ f(i)$

$tabulate(aexp, S, BV) =$
  $b = eval(bound(aexp), S, BV);$
  $f = eval(fun(aexp), S, BV);$
  $if\ not\ finite(b)\ then\ ERROR;$
  $t = alloc(b, target\_type(aexp));$
  $t[i_k] = f(i_k),\ k = 1, \ldots, n,\ where\ set(b) = \{i_1, \ldots, i_n\};$
  $return(t, b)$

[6] ? is returned when inside the bounds of an enclosing $forall$-expression.

Now we define *eval* for array expressions, both for expressions applied to an index and isolated:

$$\begin{aligned} eval(aexp\ iexp, S, BV) \quad &= \quad let\ app\_e = eval'(aexp, S, BV)\ eval(iexp, S, BV)\ in \\ &\qquad eval\_elem(app\_e, S, BV),\ app\_e \text{ closed normal form} \\ &\qquad app\_e \text{ otherwise} \end{aligned}$$

$$\begin{aligned} eval(aexp, S, BV) \quad &= \quad let\ ae = eval'(aexp, S, BV)\ in \\ &\qquad tabulate(ae, S, BV),\ ae \text{ closed normal form} \\ &\qquad ae \text{ otherwise} \end{aligned}$$

We now give semantics for the collective functions *reduce*, and *scan*, using the same kind of pseudo-code as for *eval_elem*, and *tabulate*:

$eval(reduce(bfun, aexp), S, BV) =$
  $a\ =\ eval(aexp, S, BV);$
  $b\ =\ bound(a);$
  $if\ not(finite(b))\ then\ ERROR;$
  $s\ =\ size(b);$
  $if\ s\ =\ 0\ then\ ERROR;$
  $e\ =\ enum(b);$
  $i\ =\ 0;$
  $while\ not(isDef(a[e[i]]))\ do$
    $i\ =\ i\ +\ 1;$
    $if\ i\ =\ s\ then\ ERROR;$
  $acc\ =\ a[e[i]];$
  $i\ =\ i\ +\ 1;$
  $while\ i\ <\ s\ do$
    $if\ isDef(a[e[i]])\ then\ acc\ =\ bfun(acc, a[e[i]]);$
    $i\ =\ i\ +\ 1;$
  $return\ acc$

Reduce is a function that comes in many varieties. The version defined in HERO-ML takes two arguments: an array, and a binary function that is successively applied over the array, in the order defined by the enumeration. It requires that the array is finite, and has at least one defined element: thus reducing over the empty array yields an error, as well as for an array where all elements are undefined.

This version of reduce applies the binary function sequentially. A parallel implementation could instead apply the function in a balanced tree order. This can cause discrepancies, if this function is not associative. The current version of the HERO-ML semantics allows this.

$eval(scan(bfun, aexp), S, BV) =$
  $a\ =\ eval(aexp, S, BV);$
  $b\ =\ bound(a);$
  $if\ not(finite(b))\ then\ ERROR;$
  $s\ =\ size(b);$
  $e\ =\ enum(b);$
  $i\ =\ 0;$
  $while\ not(isDef(a[e[i]]))\ do$
    $i\ =\ i\ +\ 1;$
    $if\ i\ =\ s\ then\ ERROR;$
  $t\ =\ alloc(b, target\_type(a));$
  $t[e[i]]\ =\ a[e[i]];$
  $i'\ =\ i;$
  $i\ =\ i\ +\ 1;$
  $while\ i\ <\ s\ do$
    $if\ isDef(a[e[i]])\ then$

$\sqcup$	E	A	S	D	P	$\times$
E	E	A	S	D	P	$\times$
A		A	A	A	A	A
S			S	D	P	S/P
D				D	P	$-$
P					P	P
$\times$						$\times$

$\sqcap$	E	A	S	D	P	$\times$
E	E	E	E	E	E	E
A		A	S	D	P	$\times$
S			S	S	S	S
D				D	S	$-$
P					P	S/P
$\times$						$\times$

Table 1: Result "types" of `join` ($\sqcup$), and `meet` ($\sqcap$) as a function of the argument "types" (adapted from [5]). E = `e`mpty, A = `a`ll, S = `s`parse, D = `d`ense, P = `p`redicate, $\times$ = product bound. "S/P" in the table for `j`oin means that the result is `s`parse if the product bound is finite, and a `p`redicate otherwise, and "$-$" that the combination is not allowed.

$t[e[i]] \;=\; bfun(t[e[i']], a[e[i]]);$
$i' \;=\; i;$
$i \;=\; i \;+\; 1;$
$return(t, b)$

Scan is like reduce, but rather than returning a single value (the "sum") it returns an array with the same bound as the input array, defined in the same locations, where each element holds the "partial sum" up to and including this location in the order given by the enumeration. Contrary to reduce, applying scan to the empty array is considered legal and returns the empty array.

Foreach masked array update: we give semantics for the $foreach$ statement as pseudocode that describes how the store S is updated when the statement is executed.

$foreach\ i\ in\ bnd\ do\ a\ iexp_1 \cdots iexp_n \;= exp$

Initial store S:

$e \;=\; eval(exp, S, \{i\});$
$b \;=\; eval(bnd, S, \emptyset) \sqcap eval(bound(forall\ i \;\rightarrow\; e));$
$if\ not(finite(b))\ then\ ERROR;$
$let\ \{j_1, \ldots, j_k\} \;=\; b\ in$
$tmp \;=\; alloc(b, target\_type(a));$
$for\ l \;=\; 1\ to\ k$
$\quad i_{lm} \;=\; eval(iexp_m, S[i \mapsto j_l], \emptyset)\ for\ m \;\in \{1, \ldots, n\};$
$\quad if\ i_{lm} \;\notin bound(a[i_{l1}] \cdots [i_{lm-1}])\ for\ some\ m \;\in\; \{1, \ldots, n\}\ then\ ERROR;$
$\quad tmp[l] = eval(exp, S[i \mapsto j_l], \emptyset);$
$for\ l \;=\; 1\ to\ k$
$\quad if\ isDef(tmp[l])\ then\ a[i_{l1}] \cdots [i_{ln}] = tmp[l];$

Final store: S with updated array elements according to above.

The use of the temporary array tmp is necessitated by the sequential nature of the pseudocode. Without it there would be a risk that some elements of a are overwritten before used, violating the concurrent semantics of the $foreach$ statement.

8 Semantics for Join and Meet

We now define `join` ($\sqcup$), and `meet` ($\sqcap$) for the different combinations of bounds that are possible. An overview is given in Fig. 1. Both operators are symmetric: thus, the tables are triangular. Both bounds must have the same dimensionality: consequently, `join` and `meet` of a dense bound (interval) and a product bound is not possible, since a dense bound is one-dimensional whereas a product bound always is at least two-dimensional.

In the following b stands for any bound, s for a sparse bound, $\{\, i \mid p(i) \,\}$ for a predicate bound, $l..u$ for a dense bound, and $(b_1, \ldots, b_n)$ for an n-dimensional product bound. Sparse bounds and

predicate bounds are considered to be sets, and the set operators $\cup$, $\cap$ can be applied to them. As in Section 7, $set(b)$ stands for the set defined by b. For finite sets s, $inf(s)$ is the lexicographically smallest element of s and $sup(s)$ the largest, respectively. Now, (3) - (11) defines $\sqcup$:

$$empty \sqcup b = b \tag{3}$$
$$all \sqcup b = all \tag{4}$$
$$\{\, i \mid p(i) \,\} \sqcup b = \{\, i \mid p(i) \vee i \in set(b) \,\},\ b \neq all, empty \tag{5}$$
$$s \sqcup s' = s \cup s' \tag{6}$$
$$s \sqcup l..u = min(inf(s), l)..max(sup(s), u) \tag{7}$$
$$s \sqcup b = s \cup set(b),\ b \text{ finite product bound} \tag{8}$$
$$s \sqcup b = \{\, i \mid i \in s \vee i \in set(b) \,\},\ b \text{ infinite product bound} \tag{9}$$
$$(b_1, \ldots, b_n) \sqcup (b'_1, \ldots, b'_n) = (b_1 \sqcup b'_1, \ldots, b_n \sqcup b'_n) \tag{10}$$
$$l..u \sqcup l'..u' = min(l, l')..max(u, u') \tag{11}$$

Similarly, (12) - (18) defines $\sqcap$:

$$empty \sqcap b = empty \tag{12}$$
$$all \sqcap b = b \tag{13}$$
$$s \sqcap b = s \cap set(b),\ b \neq all, empty \tag{14}$$
$$\{\, i \mid p(i) \,\} \sqcap b = set(\{\, i \mid p(i) \,\}) \cap set(b),\ b \text{ finite}, b \neq empty \tag{15}$$
$$\{\, i \mid p(i) \,\} \sqcap b = \{\, i \mid p(i) \wedge i \in set(b) \,\},\ b \text{ infinite}, b \neq all \tag{16}$$
$$(b_1, \ldots, b_n) \sqcap (b'_1, \ldots, b'_n) = (b_1 \sqcap b'_1, \ldots, b_n \sqcap b'_n) \tag{17}$$
$$l..u \sqcap l'..u' = max(l, l')..min(u, u') \tag{18}$$

A dense bound $l..u$ equals $empty$ if $l > u$.

9 How to Derive Bounds for Forall-expressions

The purpose of $forall$-expressions is to provide a convenient, and generic syntax for arrays, which is close to mathematical notation. It builds on the fact that arrays really are partial functions from indices to array values. In higher-order functional languages functions can be defined through λ-abstractions $\lambda i.e$, and the forall-expression $forall\ i \to e$ is similar except that it defines an array whose bound is derived from the syntax of e. The programmer is thus relieved from the task of explicitly defining the bounds. The implicit bounds also help making array definitions more reusable across different kinds of bounds.

The guiding principle is that the bound for $forall\ i \to e$ should safely overapproximate the domain of $\lambda i.e$, that is: $dom(\lambda i.e) \subseteq bound(forall\ i \to e)$ where $dom(f) = \{\, x \mid f(x) \neq ? \,\}$. The rationale is that if we view the function $\lambda i.e$ as the "ideal" array then, if $bound(forall\ i \to e)$ is finite, functions over $\lambda i.e$ can instead be computed over $forall\ i \to e$ by first filtering out all appearances of "?". Examples are the *reduce* and *scan* functions as defined in Section 7.

The derivation of bounds is basically a simple value analysis by abstract interpretation [3, 9]. However, contrary to a traditional static program analysis the derivation of bounds is done at run-time, every time a $forall$-expression is evaluated. Also, since arrays are not defined recursively, no widening is needed.

The bound for $forall\ i \to e$ is now given by a function $B(e, i, Y)$ where Y is a set of variables that are bound in enclosing expressions. On top level, $Y = \emptyset$. We define $B(e, i, Y)$ recursively, over the structure of e. We assume that e is a normal form (that is: all program variables have been replaced by their current values, and the resulting expression has been evaluated as far as possible). We also assume that e does not contain any free variables (implying, for instance,

that all enclosing array expressions have their array indices instantiated). Finally we assume, for simplicity, that all bound variables are distinct.

$B(e, i, Y)$ is defined by cases below. The cases are divided into three parts: first there are the "general" cases (19) – (31) that should be supported by any kind of abstract array, then there are the cases that concern multi-dimensional arrays, and finally there is a parallel read operation called *shift*, or *translation*, where elements are uniformly read from a constant offset given by a vector.

$$
\begin{aligned}
B(c, i, Y) &= \mathit{all} \quad (c \text{ constant} \neq ?) && (19)\\
B(?, i, Y) &= \mathit{empty} && (20)\\
B(y, i, Y) &= \mathit{all} \quad y \in \{i\} \cup Y && (21)\\
B(f(e_1, \ldots, e_n), i, Y) &= B(e_1, i, Y) \sqcap \cdots \sqcap B(e_n, i, Y) \quad (f \text{ ?-strict}) && (22)\\
B(\mathit{if}(e_1, e_2, e_3), i, Y) &= (B_{\mathit{true}}(e_1, i, Y) \sqcap B(e_2, i, Y)) \sqcup \\
&\quad (B_{\mathit{false}}(e_1, i, Y) \sqcap B(e_3, i, Y)) && (23)\\
B(e_1 \mid\mid e_2, i, Y) &= B_{\mathit{true}}(e_1, i, Y) \sqcup (B_{\mathit{false}}(e_1, i, Y) \sqcap B(e_2, i, Y)) && (24)\\
B(e_1 \;\&\&\; e_2, i, Y) &= B_{\mathit{false}}(e_1, i, Y) \sqcup (B_{\mathit{true}}(e_1, i, Y) \sqcap B(e_2, i, Y)) && (25)\\
B(\mathit{isDef}(e), i, Y) &= \mathit{all} && (26)\\
B(\mathit{forall}\ j \to e, i, Y) &= B(e, i, \{j\} \cup Y) && (27)\\
B(e[i], i, Y) &= \mathit{bound}(e), \quad FV(e) = \emptyset && (28)\\
B(e[i][e_1] \cdots [e_n], i, Y) &= \mathit{bound}(e), \quad FV(e) = \emptyset && (29)\\
B(a[e], i, Y) &= B(e, i, Y),\ e \neq i, FV(e) \subseteq Y && (30)
\end{aligned}
$$

$FV(e)$ stands for the set of free variables in e. For cases not explicitly covered, where $B(e, i, Y)$ still should be defined, it holds that

$$B(e, i, Y) = \mathit{all}. \tag{31}$$

B_{true}, and B_{false} are defined as follows:

$$
\begin{aligned}
B_{\mathit{true}}(\mathit{false}, i, Y) &= \mathit{empty} && (32)\\
B_{\mathit{false}}(\mathit{true}, i, Y) &= \mathit{empty} && (33)\\
B_{\mathit{true}}(\mathit{if}(e_1, e_2, e_3), i, Y) &= (B_{\mathit{true}}(e_1, i, Y) \sqcap B_{\mathit{true}}(e_2, i, Y)) \sqcup \\
&\quad (B_{\mathit{false}}(e_1, i, Y) \sqcap B_{\mathit{true}}(e_3, i, Y)) && (34)\\
B_{\mathit{false}}(\mathit{if}(e_1, e_2, e_3), i, Y) &= (B_{\mathit{true}}(e_1, i, Y) \sqcap B_{\mathit{false}}(e_2, i, Y)) \sqcup \\
&\quad (B_{\mathit{false}}(e_1, i, Y) \sqcap B_{\mathit{false}}(e_3, i, Y)) && (35)\\
B_{\mathit{true}}(e_1 \;\&\&\; e_2, i, Y) &= B_{\mathit{true}}(e_1, i, Y) \sqcap B_{\mathit{true}}(e_2, i, Y) && (36)\\
B_{\mathit{true}}(e_1 \mid\mid e_2, i, Y) &= B_{\mathit{true}}(e_1, i, Y) \sqcup (B_{\mathit{false}}(e_1, i, Y) \sqcap B_{\mathit{true}}(e_2, i, Y) && (37)\\
B_{\mathit{false}}(e_1 \;\&\&\; e_2, i, Y) &= B_{\mathit{false}}(e_1, i, Y) \sqcup (B_{\mathit{true}}(e_1, i, Y) \sqcap B_{\mathit{false}}(e_2, i, Y) && (38)\\
B_{\mathit{false}}(e_1 \mid\mid e_2, i, Y) &= B_{\mathit{false}}(e_1, i, Y) \sqcap B_{\mathit{false}}(e_2, i, Y) && (39)\\
B_{\mathit{true}}(e, i, Y) &= B_{\mathit{false}}(e, i, Y) = B(e, i, Y) \text{ for all other expressions } e && (40)
\end{aligned}
$$

Let us motivate these definitions informally. (19) is appropriate since the elements of $\mathit{forall}\ i \to c$ all will equal c, and thus should be defined for all indices i unless $c = ?$ in which case the elements should be nowhere defined as stated by (20). The elements of $\mathit{forall}\ i \to i$ will all equal i and should thus also be defined everywhere, which motivates the case in (21) where $y = i$. The other case for (21), $y \in Y$, is motivated since the variables in Y are bound in enclosing `forall`-expressions and thus should be considered constant, and thus everywhere defined across each individual evaluation.

(22) is the "elementwise application" case. It is motivated by the following: since f is strict, it holds that $f(x_1, \ldots, x_n) = ?$ as soon as some $x_j = ?$. Thus, in the rule, whenever i is outside the bounds of some e_j then $f(e_1, \ldots, e_n) = ?$. It follows that the "intersection" of the bounds for $e_1, \ldots, e_n$ will always contain all argument values for i such that $f(e_1, \ldots, e_n)$ evaluates to a value

distinct from ?. (23) is motivated by a somewhat similar argument: the *if* function is strict in its first argument (the condition), and thus $if(t_1, t_2, t_3)$ is defined only when t_1 is defined and some of t_2, t_3 are. In addition the function is "conditionally strict" in its arguments t_2 and t_3 provided that t_1 evaluates to $true$ or $false$, respectively. The somewhat complex definition, with functions B_{false}, B_{true} is exploiting this conditional strictness to obtain higher precision in the computed bound. Similar arguments motivate (24) and (25). (26) is a consequence of the fact that $isDef$ is defined (either $true$, or $false$) everywhere.

(27) keeps track of bound variables when descending into an expression. It works together with (21).

$e[i]$ can be different from ? only when $i \in bound(e)$, which motivates (28). (29) is an extension of (28) to the case of nested array application. (30), finally, is sound since evaluated arrays are ?-strict: therefore, whenever e evaluates to ? we will have $a[e] = a[?] = ?$. Thus, $a[e]$ can be defined only when e is.

The restriction on the set of free variables (FV) in some of the rules serves the purpose of delaying the calculation of bounds until enough is known to calculate a non-parametric bound. This means that possible free variables in the expressions must be instantiated first, in order to make some of the rules applicable. These free variables can, for instance, be formal arguments to some function containing `forall`-expressions. In such a case the calculation of bounds may have to be deferred until run-time, when the function is called and the formal arguments are instantiated to actual values.

9.1 Multi-Dimensional Bounds

Multi-dimensional arrays, such as matrices and tensors, are extremely important in computational applications whether it's signal or image processing, machine learning, or numerical solving of differential equations. Different data parallel and array languages have therefore developed a multitude of constructs to express operations on matrices and arrays such as selection of row or column vector from a matrix, reduction over all row/column vectors, and other operations. These constructs are however often a bit ad-hoc, often lacking generality, and the semantics can be somewhat unclear since often no precise, formal semantics is given. An important motivation for `forall`-expressions is to provide a uniform, general, and semantically well-defined syntax to express various operations on higher-dimensional arrays. Below are some examples[7]:

- `forall i-> A[i,c]` selection of column `c` from matrix `A`
- `forall j -> A[r,j]` selection of row `r` from matrix `A`
- `forall i -> A[i,i]` selection of main diagonal from matrix `A`
- `forall (i,j) -> A[j,i]` transpose of matrix `A`
- `forall i -> (forall j -> A[i,j])` conversion of matrix `A` into a nested array of arrays (row vectors)
- `forall i -> reduce(+,(forall j -> A[i,j]))` array of the sums of all rows in matrix `A`

However, if this is to be useful then these constructs must be given reasonably tight bounds. For instance, for matrix transpose the bounds in the two dimensions should be "flipped", and for the main diagonal the matrix bounds in the two dimensions should be intersected. Furthermore this should work for conventional matrix bounds as well as sparse bounds. Arrays with larger dimensionality than two should also be handled, since such arrays can turn up in applications such as, for instance, neural network processing.

First, we extend the syntax of `forall`-expressions to take tuples, representing n-dimensional vectors, as arguments:

$$forall\ (i_1, \ldots, i_n) \to e$$

[7]There should really be some figures illustrating these examples. I might add some in some later version.

We have to modify the rules to take into account that we now have tuples of bound variables. For instance, 27 will have a tuple version that looks like this:

$$B(forall\ (j_1,\ldots,j_m) \to e,(i_1,\ldots,i_n),Y) = B(e,(i_1,\ldots,i_n),\{j_1,\ldots,j_m\} \cup Y) \quad (27.1)$$

(From now on we will denote index variables by x rather than i, j.)

We now add a case to the definition of $B(e[e_1,\ldots,e_m],(x_1,\ldots,x_n),Y)$ where for some j's hold that $e_j = x_i$ for some i. $bound(e)$ will then constrain the x_i's. For simplicity we first consider the case where $bound(e)$ is a product bound $(b_1,\ldots,b_m)$: then b_j will directly constrain e_j, and thus also x_i.

9.1.1 Multi-Dimensional Product Bounds

As an example, consider $forall\ (x_1,x_2,x_3) \to a[x_2,c,x_3,x_3]$ where $bound(a) = (b_1,b_2,b_3,b_4)$. This is a three-dimensional array, defined by selecting a two-dimensional subarray of the four-dimensional array a which is then replicated in the x_1-direction. What should its bound be? Obviously x_2 should be constrained by b_1, and x_3 by both b_3 and b_4. x_1, on the other hand, should be left unconstrained. We obtain the product bound $(all, b_1, b_3 \sqcap b_4)$.

This bound can be tightened. It might be that $c \notin b_2$. Then the vector (x_2,c,x_3,x_3) will be outside the bound of a no matter what values are given to x_2 and x_3. So in this case, *empty* is a valid bound. If c is a constant then this check can be done at compile-time: otherwise the derivation of the bound will have to be deferred to when c becomes known.

We now define $B(e[e_1,\ldots,e_m],(x_1,\ldots,x_n),Y)$. We consider the case where, for each $i \in \{1,\ldots,m\}$, either $e_i = x_j$ for some j, $e_i = c_i$ for some constant c_i, or e_i is a general term such that $FV(e_i) \subseteq \{x_1,\ldots,x_n\} \cup Y$. We characterize $(e_1,\ldots,e_m)$ by (1) a partial function $p\colon \{1,\ldots,m\} \to \{1,\ldots,n\}$ such that $e_i = x_{p(i)}$ for $i \in dom(p)$, and (2) a partial function $C\colon \{1,\ldots,m\} \to Z$ such that $e_i = C(i)$ for $i \in dom(C)$. Here Z is the set of integers, and we demand that $dom(p) \cap dom(C) = \emptyset$. We define a function "$bproj_{p,C}$" that transforms a product bound $(b_1,\ldots,b_m)$ according to the following:

$$bproj_{p,C}(b_1,\ldots,b_m) = \begin{cases} (b'_1,\ldots,b'_n) & \text{if } C(i) \in b_i \text{ for all } i \in dom(C) \\ empty & \text{otherwise} \end{cases}$$

where $b'_j = \sqcap_{p(i)=j} b_i$ for $j \in \{1,\ldots,n\}$. $\sqcap_{p(i)=j} b_i$ equals all if there is no i such that $p(i) = j$. We now define

$$B(e[e_1,\ldots,e_m],(x_1,\ldots,x_n),Y) = bproj_{p,C}(bound(e)) \quad (41)$$

when

$$FV(bound(e)) = \emptyset, FV(e_1,\ldots,e_m) \subseteq \{x_1,\ldots,x_n\} \cup Y)$$

For our example $forall\ (x_1,x_2,x_3) \to a[x_2,c,x_3,x_3]$ above it holds that $p(1) = 2$, and $p(3) = p(4) = 3$. We thus have $dom(p) = \{1,3,4\}$. For C holds that $dom(C) = \{2\}$, and $C(2) = c$. If $bound(a) = (b_1,\ldots,b_4)$ then the rule set for the B function above will yield

$$B(a[x_2,c,x_3,x_3],(x_1,x_2,x_3),\emptyset) = \text{if } c \in b_2 \text{ then } (all, b_1, b_3 \sqcap b_4) \text{ else } empty$$

9.1.2 Multi-Dimensional Sparse Bounds

We now turn to the case that e has a sparse m-dimensional bound. This bound is an explicit representation of a general finite set, and thus $B(e[e_1,\ldots,e_m],(x_1,\ldots,x_n),Y)$ ought to be a sparse n-dimensional bound. At first sight it seems straightforward to define B for this kind of bound. For instance the following should hold for selecting the main diagonal, and c'th column, respectively, from the sparse matrix A:

$$\begin{array}{rcll} B(A[x,x],x,\emptyset) & = & \{\, v \mid (v,v) \in bound(A) \,\} & \text{selection of main diagonal} \\ B(A[x,c],x,\emptyset) & = & \{\, v \mid (v,c) \in bound(A) \,\} & \text{selection of column } c \end{array}$$

Both these are sparse, one-dimensional bounds.

Let's now turn to our running example. If $bound(a)$ is sparse then we ought to have the following:

$$B(a[x_2, c, x_3, x_3], (x_1, x_2, x_3), \emptyset) =$$
$$\{\, (v_1, v_2, v_3) \mid \exists (u_1, u_2, u_3, u_4) \in bound(a) \ where\ v_2 = u_1 \wedge c = u_2 \wedge v_3 = u_3 \wedge v_3 = u_4 \,\}$$

Indeed this is correct in terms of sets. However v_1 is not constrained, so this set is not finite! Obviously, this sparse bound cannot be represented by a finite set representation right away.

The problem can be dealt with in the following way. The bound can be seen as a finite set of two-dimensional vectors that are embedded into a three-dimensional space. Thus, the bound can be finitely represented by the set of 2D-vectors plus information identifying in which 2D-subspace of the 3D-space that the set resides.

To simplify the notation we use the fact that a tuple (or vector) $(v_1, \ldots, v_n)$ really is a function from $\{1, \ldots, n\}$ to some value range. A lower-dimensional vector embedded in this n-dimensional space is then a function from some subset of $\{1, \ldots, n\}$, where the embedded dimensions are given by this subset. We can use the tuple notation $(\, v_i \mid i \in dim \,)$, where dim is the set of dimensions for the subspace. In our running example we have $dim = \{2, 3\}$, and the elements in the set of 2D-vectors will be indexed as (v_2, v_3).

We now define a sparse n-dimensional bound as a triple

$$(S, dim, n)$$

where S is a finite set of $|dim|$-dimensional vectors $(\, v_i \mid i \in dim \,)$, $dim \subseteq \{1, \ldots, n\}$ is the set of dimensions in n-space in which the vectors in S are embedded, and n is the dimensionality of the bound. This bound is a finite representation of a set of n-dimensional vectors v, where there exists a $|dim|$-dimensional vector $(\, u_i \mid i \in dim \,)$ in S such that $v_i = u_i$ for all $i \in dim$. Obviously this set is infinite unless $dim = \{1, \ldots, n\}$, in which case the bound is finite and corresponds to a sparse n-dimensional bound as defined earlier.

Multi-dimensional sparse bounds can be seen as *relational databases* [8]. Such a database is essentially a set of records, where each record has a number of attributes. For a sparse bound the attributes are then the dimensions in which the vectors are embedded, and the records are the embedded vectors.

We now extend the *bproj* function to sparse bounds (S, dim, m):

$$bproj_{p,C,n}(S, dim, m) = (S', im(p), n)$$

where

$$S' = \{(\, v_j \mid j \in im(p) \,) \mid \exists (\, u_i \mid i \in dim \,) \in S. \forall i \in dim. (i \in dom(p) \implies u_i = v_{p(i)} \wedge i \in dom(C) \implies u_i = C(i))\}$$

Here, $im(p)$ stands for the *image* of p, that is: $\{\, p(i) \mid i \in dom(p) \,\}$. We require that $im(p) \subseteq \{1, \ldots, n\}$. n is typically given by the context of the bound, for instance if it is the bound for an n-dimensional forall-expression. We obtain the following version of (41), for sparse bounds $bound(e)$:

$$B(e[e_1, \ldots, e_m], (x_1, \ldots, x_n), Y) = bproj_{p,C,n}(bound(e)) \tag{42}$$

9.1.3 Join and Meet for Multi-Dimensional Sparse Bounds

In addition to the instance of the B function above, the other operations on bounds must be defined for sparse bounds. Most cases in Section 8 will still work out-of-the-box, but we need to define new versions of join and meet ($\sqcup$, $\sqcap$) for the following cases:

- both arguments are sparse multidimensional bounds, and
- one argument is a sparse multidimensional bound, and the other bound is a product bound.

These cases correspond to (6), (9), and (14) in Section 8

We now define join and meet for the first case. We first extend the *set* function to sparse bounds, mapping such a bound (S, dim, n) to the subset of Z^n that it defines:

$$set(S, dim, n) = \{\, v \in Z^n \mid \exists(\, s_i \mid i \in dim\,) \in S. \forall i \in dim. v_i = s_i \,\} \tag{43}$$

From the definition follows that *set* has the following monotonicity properties:

$$S \subseteq S' \implies set(S, dim, n) \subseteq set(S', dim, n) \tag{44}$$
$$dim \subseteq dim' \implies set(S, dim', n) \subseteq set(S, dim, n) \tag{45}$$

We want join to have the following property, for sparse bounds $\mathcal{S}_1 = (S_1, dim_1, n)$, $\mathcal{S}_2 = (S_2, dim_2, n)$:

$$set(\mathcal{S}_1) \cup set(\mathcal{S}_2) \subseteq set(\mathcal{S}_1 \sqcup \mathcal{S}_2) \tag{46}$$

This property says that join will never underestimate the sets of Z^n-vectors defined by its arguments. We now define join as follows:

$$(S_1, dim_1, n) \sqcup (S_2, dim_2, n) = (S_\sqcup, dim_\sqcup, n) \tag{47}$$

where

$$\begin{gathered} dim_\sqcup = dim_1 \cap dim_2 \\ S_\sqcup = \{\, (\, u_i \mid i \in dim_\sqcup\,) \mid u \in S_1 \cup S_2 \,\} \end{gathered} \tag{48}$$

It follows from (44), (45) that join has the desired property (46).

We now turn to meet. Similar to join (46), it should provide a safe overestimation of the intersection of the sets for its arguments:

$$set(\mathcal{S}_1) \cap set(\mathcal{S}_2) \subseteq set(\mathcal{S}_1 \sqcap \mathcal{S}_2) \tag{49}$$

We now have the following:

$$\begin{aligned} & v \in set(S_1, dim_1, n) \cap set(S_2, dim_2, n) \\ \iff\ & v \in set(S_1, dim_1, n) \wedge v \in set(S_2, dim_2, n) \\ \iff\ & \text{(by definition of } set\text{)} \\ \iff\ & (\exists s \in S_1. \forall i \in dim_1. v_i = s_i) \wedge (\exists s' \in S_2. \forall i \in dim_2. v_i = s'_i) \\ \iff\ & \text{(by simple rearrangement of the formula)} \\ \iff\ & \exists s \in S_1. \exists s' \in S_2. (\forall i \in dim_1. v_i = s_i \wedge \forall i \in dim_2. v_i = s'_i) \\ \iff\ & \exists s \in S_1. \exists s' \in S_2. (\forall i \in dim_1 \cup dim_2. v_i = merge(s, s')_i) \end{aligned} \tag{50}$$

where

$$merge(s, s')_i = \begin{cases} s_i & \text{if } i \in dim_1 \\ s'_i & \text{if } i \in dim_2 \end{cases}$$

$merge(s, s')$ is well-defined iff $s_i = s'_i$ for all $i \in dim_1 \cap dim_2$. We now define meet, viz.

$$(S_1, dim_1, n) \sqcap (S_2, dim_2, n) = (S_\sqcap, dim_\sqcap, n) \tag{51}$$

where

$$\begin{gathered} dim_\sqcap = dim_1 \cup dim_2 \\ S_\sqcap = \{\, merge(s, s') \mid s \in S_1, s' \in S_2 \text{ where } s_i = s'_i \text{ for all } i \in dim_1 \cap dim_2 \,\} \end{gathered} \tag{52}$$

It now follows from (50), (51), and (52) that $v \in set(S_1, dim_1, n) \sqcap (S_2, dim_2, n)$ precisely when $v \in set(S_1, dim_1, n) \cap set(S_2, dim_2, n)$, that is: (49) is fulfilled and meet furthermore provides a tight bound.

We finally define join and meet between a sparse, n-dimensional bound (S, dim, n), and an n-dimensional product bound $(b_1, \ldots, b_n)$. Let $\circ$ be any of $\sqcup$, $\sqcap$. Then:

$$(S, dim, n) \circ (b_1, \ldots, b_n) = (b'_1 \circ b_1, \ldots, b'_n \circ b_n) \tag{53}$$

where

$$b'_i = \begin{cases} all & \text{if } i \notin dim \\ (\{\, x \mid \exists s \in S.x = s_i \,\}, \{1\}, 1) & \text{if } i \in dim \end{cases}$$

Thus, b'_i is the projection of S in dimension i. It is not hard to show that $(b'_1, \ldots, b'_n)$ provides a safe approximation of (S, dim, n).

9.1.4 Multi-Dimensional Predicate Bounds

Predicate bounds are in a way similar to sparse bounds, but can be represented much more directly by executable functions since they are not finite. This simplifies the derivation of multi-dimensional predicate bounds considerably. For our running example $forall(x_1, x_2, x_3) \to a[x_2, c, x_3, x_3]$, if b is a predicate bound then we obtain

$$B(a[x_2, c, x_3, x_3], (x_1, x_2, x_3), \emptyset) = \{(x_1, x_2, x_3) :\ member((x_2, c, x_3, x_3), bound(a))\}$$

It is straightforward to give a general definition of the B function for predicate bounds:

$$B(e[e_1, \ldots, e_m], (x_1, \ldots, x_n), Y) = \{(x_1, \ldots, x_n) :\ member((e_1, \ldots, e_m), bound(e))\} \tag{54}$$

(54) is generic: it works for general index expressions e_i. In particular this includes the extended index expressions considered in Section 9.2 below.

9.2 Bounds for a Class of Linearly Shifted Arrays

We now extend the class of expressions, where the "B" function can give a tight bound, to expressions $e[e_1, \ldots, e_m]$, where $e_i = s_i \cdot x_{p(i)} + o_i$, with $s_i \neq 0$ and $o_i \in Z$ whenever $i \in dom(p)$. Note that this includes the previously considered expressions $e_i = x_{p(i)}$ whenever $s_i = 1$, and $o_i = 0$. We will refer to s_i as a *stride*, o_i as an *offset*, and to arrays accessed with index expressions as above as *stride-shifted arrays*.

Here are three examples of what the added expressiveness can be used for:

- `forall i -> A[-i]` reversal of `A`
- `forall (i,j) -> A[i-1,j-1]` shift of matrix `A` with vector `(1,1)`
- `forall i -> (A[2*i] + A[2*i+1])` addition of of the neighboring evenly- and oddly-indexed elements of `A`

All these are examples of *get communication*, or *parallel read*, where an imaginary processor at coordinate `i` (or `i,j`) reads an element of `A` from position `-i` (or `i-1,j-1`, or `2*i` and `2*i+1`). Such communication is common in data parallel algorithms: in particular shift operations are common and they often have efficient hardware implementations. The general format for parallel read is `forall i -> A[g(i)]`, where the `g(i)`, for `g(i)` $\in$ `bound(A)`, constitute the *source addresses* of the parallel read.

How can tight bounds be found for arrays created by parallel read? There is no general answer – the problem is undecidable in general – but some clues can be found by studying the domains of the corresponding functions. For functions A and g where g is injective, and thus has a left inverse $g^{-1}\colon im(g) \to dom(g)$, we have:

$$\begin{aligned} dom(\lambda i.A(g(i))) &= \{\, i \mid g(i) \in dom(A) \,\} \\ &= \{\, i \mid g^{-1}(g(i)) \in g^{-1}(dom(A)) \,\} \\ &= \{\, i \mid i \in g^{-1}(dom(A)) \,\} \\ &= g^{-1}(dom(A)) \end{aligned}$$

This fact can often be used to find functions that translate bounds according to g^{-1}. For the parallel read considered here, we have a function $g(x) = s \cdot x + o$, and thus $g^{-1}(y) = (y - o)/s$. We now define a function "ss" ("stride-shift") that takes a stride s, an offset o, and a one-dimensional bound b, and returns a bound representing $g^{-1}(b)$:

$$
\begin{aligned}
ss(all, s, o) &= all && (55)\\
ss(empty, s, o) &= empty && (56)\\
ss(l..u, s, o) &= \lceil (l-o)/s \rceil .. \lfloor (u-o)/s \rfloor, \quad s > 0 \text{ (dense bound 1)} && (57)\\
ss(l..u, s, o) &= \lceil (u-o)/s \rceil .. \lfloor (l-o)/s \rfloor, \quad s < 0 \text{ (dense bound 2)} && (58)\\
ss(\{i_1, \ldots, i_n\}, s, o) &= \{\, (i_j - o)/s \mid s \text{ divides } i_j - o \,\} \text{ (finite sparse 1D bound)} && (59)\\
ss(\{i : p(i)\}, s, o) &= \{i : p(s \cdot i + o)\} \quad \text{(predicate bound)} && (60)
\end{aligned}
$$

9.2.1 Bounds for One-dimensional Shifted Arrays

We now define more precise bounds for stride-shifted one-dimensional arrays:

$$B(e[s \cdot i + o], i, Y) = ss(bound(e), s, o), \quad FV(e) = \emptyset \tag{61}$$

This rule supersedes (28), which becomes a special case with $s = 1$ and $o = 0$. A similar rule can be formulated for nested arrays, which in the same way supersedes (29).

9.2.2 Multi-dimensional Product Bounds

We now extend the bounds for shifted arrays to the multi-dimensional case, by extending the B function to expressions $e[e_1, \ldots, e_m]$ where $e_i = s_i \cdot x_{p(i)} + o_i$ for $i \in dom(p)$. We first treat the case where e has a product bound $(b_1, \ldots, b_m)$, and we define an extended version $bproj_{p,C,s,o}$ of the "$bproj$" function in Section 9.1.1, with additional parameters $s = (\, s_i \mid i \in dom(p) \,)$ and $o = (\, o_i \mid i \in dom(p) \,)$, respectively, that define the strides and offsets s_i, o_i for the occurrences of $x_{p(i)}$:

$$bproj_{p,C,s,o}(b_1, \ldots, b_m) = \begin{cases} (b'_1, \ldots, b'_n) & \text{if } C(i) \in b_i \text{ for all } i \in dom(C) \\ empty & \text{otherwise} \end{cases}$$

where $b'_j = \sqcap_{p(i)=j} ss(b_i, s_i, o_i)$ for $j \in \{1, \ldots, n\}$. The new version of $bproj$ is now used to define the "B" function for multi-dimensional product bounds:

$$B(e[e_1, \ldots, e_m], (x_1, \ldots, x_n), Y) = bproj_{p,C,s,o}(bound(e)) \tag{62}$$

when $FV(bound(e)) = \emptyset, FV(e_1, \ldots, e_m) \subseteq \{x_1, \ldots, x_n\} \cup Y$. This definition supersedes (41).

9.2.3 Sparse Multi-Dimensional Bounds

We now define a version of $bproj$ for sparse multi-dimensional bounds:

$$bproj_{p,C,s,o,n}(S, dim, m) = (S', im(p), n)$$

where

$$S' = \{(\, v_j \mid j \in im(p) \,) \mid \exists (\, u_i \mid i \in dim \,) \in S. \forall i \in dim. (i \in dom(p) \implies u_i = s_i \cdot v_{p(i)} + o_i \wedge i \in dom(C) \implies u_i = C(i))\}$$

We obtain the following version of (42), for stride-shifted arrays with sparse bounds $bound(e)$:

$$B(e[e_1, \ldots, e_m], (x_1, \ldots, x_n), Y) = bproj_{p,C,s,o,n}(bound(e)) \tag{63}$$

This definition supersedes (42).

9.2.4 Detecting Index Expressions for Stride-Shifted Arrays

In this section we consider index expressions that are on the format $s \cdot x + o$. However there are many other index expressions that are semantically equal, but have different syntax. Naïve pattern-matching on the syntax will miss many such cases: for instance the expression $-2 \cdot x + 5 + 3 \cdot x$ has the wrong syntax, and will not be recognized.

HERO-ML therefore has a set of transformations that transform index expressions into the prescribed format, if possible, in order to detect semantically equivalent index expressions. For instance the expression

$$A[-3 \cdot i + 27 + 2 \cdot i + (14 - 14) \cdot i \cdot i]$$

will be transformed into

$$A[-i + 27],$$

which is on the prescribed format. The exact rules how to transform index expressions are given in Appendix A.

10 Possible Future Extensions of HERO-ML

In this section we give a "wish list" for a number of possible future extensions of HERO-ML, including simple syntactic conveniences as well as more substantial additions.

10.1 User-Defined Functions and Procedures

HERO-ML in its current form is an experimental language, for research purposes, rather than a production language. The focus is thus on abstract arrays rather than standard language constructs. Therefore the current version does not have user-defined functions and procedures. They are however straightforward to add, and a full-fledged production version of HERO-ML should have them.

10.2 Elemental Overloading, and Promotion of Scalars

Many array languages offer a syntactic convenience known as *elemental intrinsics overloading*. This refers to the overloading of "scalar" functions or operators to work also elementwise on arrays, for instance to let `a + b` stand for the elementwise addition of the arrays `a` and `b`. A related syntax is to "promote" scalars into constant-valued arrays when the scalar appears in a position where an array is expected. this kind of syntax is often highly appreciated, but is mostly offered only for a restricted set of operators in an ad-hoc fashion.

In a language with `forall`-expressions and explicit typing, general rules can be given for how to resolve this kind of overloading. Here are some examples, where "`+`" is addition of floats, `a` and `b` are abstract arrays with elements of type float, and `source` is an abstract array with elements of the same type as the index type for `a`:

- `a + b` → `forall i -> a[i] + b[i]` (elemental overloading)
- `a + 1.0` → `forall i -> a[i] + 1.0` (elemental overloading with constant)
- `a[source]` → `forall i -> a[source[i]]` (overloading of get communication)
- `1.0 | 1..10` → `(forall i -> 1.0) | 1..10` (promotion of `1.0`, followed by explicit restriction, to create a constant array with bound `1..10`)

Some of this can be done in Haskell, for operators and functions belonging to some predefined class [6]. But there are ways to do this for operators and functions not necessarily restricted to some class or similar. In [11, 12] it is shown how to do this systematically for explicitly typed languages, formulating the problem as a combined type inference and code transformation system that resolves the overloading during th type checking.

10.3 Array Syntax for Selection of Substructures

Another popular feature in array languages is to offer convenient notation for selecting substructures. For instance `a[i,*]` might refer to the `i`'th row of the matrix `a`, and `a[*,k]` to the `k`'th column. This syntactic sugar can easily be resolved using `forall`:

- `a[i,*]` → `forall k -> a[i,k]` (selection of row)
- `a[*,k]` → `forall i -> a[i,k]` (selection of column)

10.4 Other Syntactical Conveniences

Many other syntactical conveniences are possible. For instance we might let

```
foreach i do x[iexp] = exp
```

stand for

```
foreach i in all do x[iexp] = exp
```

Such simple syntactic short-hand forms can increase the readability quite a bit.

10.5 More General Index Types

HERO-ML has integers, and tuples of integers, as index types. But also other index types could be considered. The only properties required are that values of the type can be compared for equality, and that there is a total ordering on them. This would allow abstract arrays that are basically maps, enabling data parallel programming with maps.

10.6 A Richer Set of Numerical Types

In numerical computing the results are typically approximate. At the same time, many embedded numerical applications are resource-demanding. Resources can be saved by lowering the numerical precision, creating a tradeoff between cost and precision. A prominent example is deep neural networks where the cost sometimes can be significantly reduced, while maintaining acceptable precision, by reducing the word length for floating-point operations or swithhing from floating point arithmetic to fixed point arithmetic. A richer set of numerical types would thus leverageee HERO-ML as a tool for optimizing the numerical precision in modeled appplications.

10.7 User-defined Bounds

Bounds are set representations. We require that certain functions and operations on bounds are supported, and that they have certain properties. These could be seen as an abstract interface, hiding the internal details of the set representations. An intriguing possibility is to provide means for defining user-defined bounds that implement the interface.

11 Reference Implementation

A prototype interpreter has been implemented for the language- It can parse and execute HERO-ML programs entirely in software, supporting the complete language as specified in this document. This allows programs to be quickly validated and debugged without the need to first translate the code into some auxiliary program format, to be compiled and run separately. The interpreter is a command line application written entirely in F#, and so should be supported on all major platforms without modifications to the source code.

The overall design of the interpreter is fairly straightforward. Taking the name of a HERO-ML source file as a command line argument, it begins by parsing this file to generate an abstract syntax tree (AST) for the program. After this, the AST undergoes a validation step where any remaining static correctness checks that could not practically be performed during parsing (such

as type checking) are carried out. Finally, if the program passed the previous two steps, it is executed from start to finish using the AST as the program format.

Throughout the program execution, a representation of the current HERO-ML program state is maintained in memory in the form of a dictionary data structure (implemented using the type *Map* from the standard library of F#), which keeps an entry for each program variable holding the variable's current runtime value. Variables that are local to, e.g., forall expressions are also added to the dictionary, but only temporarily while the expression in which they are bound is being evaluated. The actual program execution is an iterative process analogous to applying the semantic rules of Section 7 to the AST nodes and the program state in a repeated fashion, generating updated program states until the program terminates. Any *out* statements occurring in the program are handled by printing a textual representation of the specified output value to the console.

To represent the runtime values of variables as well as any intermediate results generated during expression evaluations, a custom type $Value$ is used, which is an F# discriminated union with one case for each HERO-ML type. The scalar types of HERO-ML—integers, floats, and booleans—are represented using the corresponding primitive data types of F#. Bounds and arrays are represented using custom aggregate types, described in more detail below. In addition, the $Value$ union includes cases for certain symbolic values, such as the special value "?", as well as an "ERROR" value which symbolizes a program error generated during expression evaluation. In most cases, occurrences of the ERROR value immediately cause execution to terminate with an error message printed to the console.

The runtime representation used for bounds is fairly simple, using an F# discriminated union for the different types of bounds: dense, sparse, predicate, and product bounds, as well as *empty* and *all*. Some of these bounds require additional parameters for their definition: A dense bound $l..u$ is given by a pair of integers, and a product bound $(b_1, \ldots, b_m)$ stores the factors $b_1, \ldots, b_m$ in an F# list. For a sparse bound (S, dim, n) (see Section 9.1.2), the set S uses the Set datatype from the F# standard library, as this type provides fast operations such as membership lookups and computing set unions and intersections. The individual (one- or multidimensional) indices of S are represented using a custom type $Index$, which is simply a type alias for an integer list. For predicate bounds, the predicate is stored as an F# function of type $Index \rightarrow bool$ which, when applied to an index value, evaluates the predicate with the local index variable(s) of the predicate temporarily bound to the given index.

To represent abstract arrays, two pieces of data are combined: a bound (of the type described above) as well as a data structure which maps individual indices of the bound to their corresponding array elements. Once an array has been defined, the operation of accessing its elements by using indices involves two steps: First the index is checked for membership in the bound. If this check fails, then the operation immediately returns the symbolic ERROR value to signal an access out-of-bounds. Otherwise, the mapping component of the array is used to look up the value stored for the index. The mapping component of the array definition comes in an implicit as well as an explicit variant. The implicit representation is used for array comprehensions and forall expressions before they have been evaluated into the tabular form (see Section 7). It implements the element lookup step using an F# function of type $Index \rightarrow Value$, which is defined to evaluate the body expression of the array with the index variable(s) temporarily bound to the query index. It should be noted that although the index has been confirmed to belong to the bound of the array prior to the lookup, the evaluation of the body might yet result in the symbolic ERROR value. This can occur if the array is a forall expression and its bound is a strict overestimation of the true domain of the body expression. Since the language semantics states that in this case, the overall result of the array access should be "?" instead of it generating an error, the ERROR value is simply replaced by the symbolic value representing "?".

The explicit variant is used for any arrays that are in the tabular form, which includes explicitly defined arrays as well as implicitly defined arrays that have undergone evaluation (note that this implies the arrays are finite). This representation stores every element of the array in fully evaluated form in a "data store", which is a one-dimensional F# array with the same number of elements as the HERO-ML array being represented, with the elements ordered by the lexicograph-

ical ordering of their indices. This provides a memory-efficient representation which supports fast element lookups (constant time for one-dimensional dense arrays). In the case of multidimensional HERO-ML arrays, the query index must first be translated into a one-dimensional index for the lookup to be performed. This is done as follows. Upon creation of an n-dimensional finite array with bound b, the minimum bounding hyperrectangle of b, $B = (l_1..u_1, \ldots, l_n..u_n)$, is calculated (where $B = b$ if b is dense). The lexicographical ordering of n-dimensional indices then gives a bijective map from B to the range $0, \ldots, size(B)-1$, where $size(B) = w_1 \times \cdots \times w_n, w_j = u_j - l_j + 1$. Specifically, given an index $(i_1, \ldots, i_n) \in B$, its (0-based) position in the lexicographical order is given by

$$i' = (i_1 - l_1) \times (w_2 \times w_3 \times \cdots \times w_n) + (i_2 - l_2) \times (w_3 \times w_4 \times \cdots \times w_n) + \cdots + (i_n - l_n)$$

This operation could be viewed as "unfolding" the n-dimensional box B into a one-dimensional range of indices. Conversely, given an unfolded index $i' \in [0, size(B) - 1]$, the corresponding "folded" index $(i_1, \ldots, i_n) \in B$ is given by

$$\begin{aligned} i_1 &= l_1 + (\lfloor i'/(w_2 \times w_3 \times \cdots \times w_n) \rfloor \bmod w_1) \\ i_2 &= l_2 + (\lfloor i'/(w_3 \times w_4 \times \cdots \times w_n) \rfloor \bmod w_2) \\ &\vdots \\ i_n &= l_n + (i' \bmod w_n) \end{aligned}$$

If the array is dense (i.e., if its bound b is), then the one-dimensional index into the data store corresponding to a particular n-dimensional index is given simply by the above unfolding operation, which takes $\Theta(n)$ time to perform. However, if the array is sparse with $set(b) \subsetneq set(B)$, then this means that there will be elements which are adjacent in the data store although their (unfolded) indices differ by more than 1, as the intervening indices correspond to n-dimensional indices that fall inside B but are not part of b itself. Thus, in this case the data store forms a sequence of dense clusters of elements—of elements at consecutive indices (or consisting of single isolated elements)—To facilitate element lookups in this case, a type of cluster dictionary is kept alongside the data store. The cluster dictionary is an array of integer triples (s_j, o_j, m_j), where s_j gives the unfolded index of the first element of cluster j, o_j gives the offset of where the cluster begins in the data store, and m_j gives the size of the cluster. In other words, cluster j spans the unfolded indices $s_j, \ldots, s_j + m_j - 1$ and its elements are located at positions $o_j, \ldots, o_j + m_j - 1$ in the data store. The cluster dictionary is sorted on the starting index s_j (or, equivalently, the offset o_j), allowing the cluster into which an unfolded index i' falls to be found quickly by a binary search for the smallest value j' such that $s_{j'} \leq i'$. The element corresponding to i' is then located at position $o_{j'} + i' - s_{j'}$ in the data store. As the number of clusters in an array of m elements is $O(m)$, each element lookup thus takes $\Theta(n) + O(\log m)$ time in total, and the memory requirements are $\Theta(m)$.

References

[1] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, Apr. 1994.

[2] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, 1998.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Jan. 1977.

[4] J. Holmerin. Implementing data fields in Haskell. Technical Report TRITA-IT R 99:04, Dept. of Teleinformatics, KTH, Stockholm, Nov. 1999. http://www.es.mdh.se/publications/5438-.

[5] J. Holmerin and B. Lisper. Data Field Haskell. In G. Hutton, editor, *Proc. Fourth Haskell Workshop*, pages 106–117, Montreal, Canada, Sept. 2000.

[6] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, USA, 2000.

[7] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1994.

[8] D. Maier. *The Theory of Relational Databases*. Pitman, London, 1983.

[9] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2^{nd} edition*. Springer, 2005.

[10] Thinking Machines Corporation, Cambridge, MA. *Getting Started in *Lisp*, June 1991.

[11] C. Thornberg. *Towards Polymorphic Type Inference with Elemental Function Overloading*. Licentiate thesis, Dept. of Teleinformatics, KTH, Stockholm, May 1999. Research Report TRITA-IT R 99:03.

[12] C. Thornberg and B. Lisper. Elemental function overloading in explicitly typed languages. In M. Mohnen and P. Koopman, editors, *Proc. 12th International Workshop of Implementation of Functional Languages*, pages 31–46, Aachen, Germany, Sept. 2000.

A Detecting Index Expressions for Stride-Shifted Arrays

We now describe or current method how to detect that an index expression is on a format equivalent to either $s \cdot x + o$, where x is an index variable and s, c are integers, or to a constant. We use this method to decide, for each index expression e_i in an array application $e[e_1, \ldots, e_m]$, whether it equals $s_i \cdot x_j + o_i$ for some $forall$ variable x_j, where $s_i \neq 0$, or if it equals a constant. The function L, defined below, takes an evaluated index expression as argument and tries to rewrite it into the format defined above.

This information defines the bounds for multi-dimensional $forall$-expressions through the different "*bproj*" functions, as described in Sections 9.1 and 9.2. The *bproj* functions are in turn defined by the partial functions C and p, which are defined as the smallest partial functions satisfying the following:

$$L(eval(e_i)) = c_i \quad \Longrightarrow \quad C(i) = c_i \tag{64}$$

$$L(eval(e_i)) = s_i \cdot x_j + o_i \text{ where } s_i \neq 0 \quad \Longrightarrow \quad p(i) = j \tag{65}$$

The L function is defined below. In each rule c, s, and o are integers, and x is a variable:

$$L(c) = c \tag{66}$$

$$L(x) = x \tag{67}$$

$$L(-e) = (-s) \cdot x + (-o), \quad L(e) = s \cdot x + o \tag{68}$$

$$L(e_1 \cdot e_2) = L(e_2 \cdot e_1) = 0, \quad L(e_1) = 0 \tag{69}$$

$$L(e_1 \cdot e_2) = L(e_2 \cdot e_1) = (c \cdot s) \cdot x + (c \cdot o), \quad L(e_1) = s \cdot x + o, \quad L(e_2) = c \tag{70}$$

$$L(e_1 + e_2) = L(e_2 + e_1) = s \cdot x + (o + c), \quad L(e_1) = s \cdot x + o, \quad L(e_2) = c \tag{71}$$

$$L(e_1 + e_2) = L(e_2 + e_1) = (s_1 + s_2) \cdot x + (o_1 + o_2), \quad L(e_k) = s_k \cdot x + o_k \text{ for } k = 1, 2 \tag{72}$$

$$L(e_1 - e_2) = L(e_1 + (-e_2)) \tag{73}$$

$$L(e) = e, \quad e \text{ is any other type of expression} \tag{74}$$

The rules are not complete. There are cases where they will fail to rewrite expressions into the required format even when they should be able to. But we believe that these cases will be very rare in practice.

An alternative method would be to attempt to rewrite the index expression into polynomial standard form, for which there are well-known methods. It can easily be decided from the standard form whether the expression has the right format. We might implement this method in the future.