Equelle Reference Manual

Atgeirr Flø Rasmussen, SINTEF ICT

Contents

1 Introduction

Equelle is a domain-specific language intended for writing simulators, in the sense of programs that numerically solve partial differential equations. The language was created at SINTEF ICT, department of Applied Mathematics, starting in 2013.

1.1 Motivation

The primary motivation for Equelle is to separate the concerns of the application expert who knows a field, its physics and equations from the computational scientist who knows parallel programming and linear solvers.

We also seek to minimize the work needed to port simulators to new architectures and hardware types. This is achieved by allowing multiple back-ends to the language, reducing the question of supporting a new hardware type to implementing a new back-end.

We have chosen not to start at the level of the PDEs themselves. Instead, our intention is that Equelle should make it easy to express the discrete equations for a simulator. Furthermore, we have concentrated on features useful for finite volume methods, as such methods are our primary focus.

1.2 Basic syntax

An Equelle program consists of a sequence of statements. Statements are terminated by newlines, unless preceded by an ellipsis (three periods, `...`). Comments starts with the hash character `#` and run to the end of the line, and an ellipsis does not make the comment continue to the next line. Blocks such as function bodies or `For` loop bodies are delimited by curly brackets ({ }).

1.3 Vectorized notation

In languages like C or Fortran, numerical programs usually have a large number of loops. In Equelle we seek to have fewer loops, and especially to avoid indexing. Indexing is a major source of errors in numerical computation, and we have sought to minimize them by using vectorized notation. In

Equelle, the statement `a = b + c*d` can just as well refer to collections of number as to single numbers. All such statements are interpreted element-wise, similar to Matlab's initial-point operators such as `.*` or `./` which do not perform matrix operations but element-wise operations.

1.4 The grid concept in Equelle

In any Equelle program, a computational grid is assumed to exist. It is constant and unchanging from the view of the program, and it is up to the back-end to actually create or read a particular grid, when the simulator is run.

Grids are cell complexes. They consist of entities of varying dimensionality. The entities of highest dimension are called cells and all other entities can be obtained as intersections of cells. The converse also holds: all possible intersections of cells are entities of the grid. This means that Equelle grids are unstructured, and capable of representing almost arbitrary computational grids. It also means that there is no notion of non-conforming grid, since all cell-cell intersections are by definition entities of the grid.

1.5 The `Collection Of` types

In most programming languages, a collection of data will be accessed either as a sequence, or by random access, using indices. In Equelle, data will often be associated with a subset of grid entities instead. A variable can be created that has one element per cell in the grid, for example. The Equelle compiler ensures that such a variable cannot be used in a context where it does not make sense, such as adding it to a variable containing one element per face. An Equelle `Collection` always has a domain, and can be thought of as a 1-1 mapping from that domain to the data. In traditional languages, such mappings would implicitly be from the integer indices to the data. New domains can be created from user input, to capture concepts such as applying a certain boundary condition to only part of the boundary, or using different equations on part of the grid. There is also a set of operators (`On`, `Extend`) that manipulates these mappings, giving a great deal of flexibility without the risk of indexing errors. The `On` concept is described in more detail in section 2.3.

1.6 Type checking and type inference

Equelle is strongly typed. Making a mistake in typing will usually cause a compile error. This is a safety feature, and when combined with the domains of collections being part of its type it is a strong safeguard against errors that are easy to make in other languages.

Mostly, Equelle can be said to be statically typed. However, the domains of `Collection`s can be dynamically computed or input, so that part of the type system can be said to be dynamic. However, this is all checked at compile time.

1.7 Declaration and assignment, immutability

Unlike languages like C++ (at least before `auto` was allowed), the user is usually not burdened with declaring the types of variables. The Equelle compiler is able to infer the correct types of any valid expression. Therefore Equelle statements can be very concise, yet unlike languages such as Python or Matlab, type safety is enforced at compile time.

To declare a variable, use a single statement with a colon separating the variable name and its type:

```
volume : Scalar
```

To assign a variable, use the `=` sign:

```
volume = 5.67
```

This can be combined, as follows:

```
volume : Scalar = 5.67
```

The declarations are entirely optional. They can be included as a form of documentation, or left out according to the wishes of the programmer. There is an important restriction on variable and function names: user-declared names must start with a lower case letter. Names starting with a capital letter are reserved for Equelle. In the remainder of the name, any letter or number can be used, or underscores.

By default, Equelle variables are immutable, and can only be assigned to once. This makes it easier for the compiler and back-end to parallelize

Type	*Semantics*
`Scalar`	A single floating-point number
`Vector`	A pair or triple of `Scalar`s (depending on grid dimension)
`Bool`	A flag that is `True` or `False`
`Cell`	A cell entity of the grid
`Face`	A face entity of the grid
`Edge`	An edge entity of the grid
`Vertex`	A vertex entity of the grid

Table 1: Basic types in Equelle

programs correctly. It is possible to create mutable variables, but it must be done explicitly, but declaring the variable to be `Mutable`.

2 Types in Equelle

2.1 Basic types

There are 7 basic types in Equelle, as shown in Table 1. Note that there is no separate type for integer numbers. This helps avoiding some types of errors that are widespread in other languages, for example the expression `1/2` will yield an expression of type `Scalar` with the value 0.5, and not zero as in C.

Vectors are special in that the number of elements will depend on the number of grid dimensions. A simulator can be created (depending on the back-end used) that is capable of running simulations on both 2D and 3D grids, and the Vector class will be appropriately sized in both cases. One can obtain the Scalar elements of a Vector by indexing (starting from 0):

```
a : Vector
...
b : Scalar = a[0]    # First ('x') element.
c : Scalar = a[2]    # Third ('z') element.
```

Use of vector indexing is checked, and a program that uses the index 2 like in the assignment of c above, will impose the restriction on the resulting simulator that it requires 3 grid dimensions.

2.2 Collections

Variable can contain several elements of the same basic type. Such a variable is called a collection. To declare a collection you use `Collection Of` followed by a basic type, the keyword `On` and an expression for the domain of the collection:

```
a : Collection Of Scalar On AllFaces()
b : Collection Of Cell On InteriorFaces()
c : Collection Of Vector On AllCells()
```

Arithmetic operators can be applied only between collections that are `On` the same domain.

2.3 Domains and the `On` concept

A domain is defined to be a set of unique grid entities (no repeats), all of the same dimension. Several basic domains are available via built-in Equelle functions, such as `AllCells()`, `BoundaryFaces()` and `InteriorVertices()`. It is also possible to specify domains dynamically, to be set from user input when running a simulation. For example (including optional type declaration):

```
dirichlet_boundary : Collection Of Face Subset Of AllFaces()
dirichlet_boundary =  InputDomainSubsetOf(AllFaces())
```

This new domain can be used in all ways that the basic built-in domains can. Note that the `Subset Of` part will be discussed in section 2.4.

Given a domain, collections can be declared to be `On` that domain. Such a collection can be thought of as a 1-1 mapping from the domain to the data of the collection. There is no ordering implicit in this. Consider the following declaration:

```
a : Collection Of Scalar On BoundaryFaces()
```

This can be viewed as mapping taking $f \in$ `BoundaryFaces()` $\rightarrow a(f)$.

Even if on a low level there exist some ordering of the boundary faces of the grid, it is not needed by nor accessible from the Equelle program. However, if a different collection b was declared to be `On` the same domain, it would be possible to add or multiply the collections, and the result would also be

`On BoundaryFaces()`, for example can $a + b$ be viewed as mapping taking $f \in$ `BoundaryFaces()` $\rightarrow a(f) + b(f)$.

Finally, note that not all collections of grid entities are domains. For example the collection returned from `FirstCell(InteriorFaces())` can possibly contain repeated instances of one or more cells.

2.4 The `Subset Of` concept

It is often necessary to extend or restrict collections to larger or smaller domains. Therefore it is important to know for any domain, what other domains it is contained in. For the built-in domains this is obvious and given. For example `BoundaryCells()` is contained within `AllCells()`. For domains that are input from the user, the containing domain must be given as an argument to the `InputDomainSubsetOf()` function, as seen in the `dirichlet_boundary` example of 2.3.

Given a collection on some domain, one may extend the collection to larger domains by padding it with zeros. This is only possible for collections of `Scalar` type. To do this one can use the `Extend` operator. To continue the previous example:

```
x : Collection Of Scalar On dirichlet_boundary
...
y : Collection Of Scalar On BoundaryFaces()
y = x Extend BoundaryFaces()
```

Similarly, one may do the opposite and restrict a collection to a subdomain by using `On` as an operator:

```
x : Collection Of Scalar On BoundaryFaces()
...
y : Collection Of Scalar On dirichlet_boundary
y = x On dirichlet_boundary
```

A user-specified domain may or may not be a subset of another user-specified domain, but that is not possible to decide at compile time, and so you can never extend or restrict a collection from one such domain to another with `Extend` and `On` directly. However, it is possible to do so via a containing domain:

```
d1 : Collection Of Cell Subset Of AllCells()
```

```
d2 : Collection Of Cell Subset Of AllCells()
x1 : Collection Of Scalar On d1
...
x2 = x1 Extend d2                    # Error
x2 = x1 On d2                        # Error
x2 = ((x1 Extend AllCells()) On d2)  # Ok.
```

The above example first extends x1 with zeros to `AllCells()` and then restricts it to d2. If d2 overlaps d1, then x2 will have the same value as x1 for the cells of d2 that are also in d1, and zero for the cells that don't.

There is one way the use of `On` as an operator is very different from the use of `Extend`: the collection on the right hand side of the `On` does not have to be a domain. Still, all elements of the collection have to be in the domain of the left hand side. This is looked at in more detail in section 3.5.

2.5 Sequences

Unlike a `Collection`, that is a mapping from some grid domain, a `Sequence` is a mapping from the integers. It is quite limited what a sequence can be used for, currently the only usage is to specify the data over which a `For` loop iterates. The declaration syntax for sequences is similar to that for collections, and as with collections it is possible to have sequences of basic types only. Example usage:

```
timesteps : Sequence Of Scalar
timesteps = InputSequenceOfScalar("timesteps")
For dt In timesteps {
    ...
}
```

The above loop is guaranteed to execute in sequential order, with `dt` taking one by one the values in the sequence `timesteps`.

2.6 Arrays

First, note that the `Array` type is not intended for long arrays. That would be handled by a `Collection` in Equelle. Rather, it is meant to collect small sets of variables for convenience, especially as function arguments or return types. The size of an array must be given in the program. It is not possible

to use arithmetic operators with arrays, the only allowed operation is the indexing operation that obtains an element from the array. Indexing starts from zero. Square brackets can be used to construct an array. Example usage:

```
bothFunc : Function() ...
           -> Array Of 2 Collection Of Cell On InteriorFaces()
bothFunc() = {
    -> [FirstCell(InteriorFaces()), SecondCell(InteriorFaces())]
}
both = bothFunc()
first = both[0]
second = both[1]
wrong = both[2]  # This generates a compiler error
```

As with access to `Vector` types, indexed access is checked by the compiler. If an expression is an `Array` of either `Vector` or `Collection Of Vector`, the first indexing operator always refers to elements of the `Array`.

2.7 Function types

It is a goal to eventually support very flexible function templates in Equelle. However for now, user-defined functions must have their type declared, meaning that the types of arguments and the return type must be given. Example usage:

```
compRes : Function(u : Collection Of Scalar On AllCells(), ...
                   u0: Collection Of Scalar On AllCells(), ...
                   dt : Scalar) ...
                   -> Collection Of Scalar On AllCells()
```

Some built-in functions have some polymorphism, such as `InputDomainSubsetOf`, `Centroid` or `Output`. There is at the moment no Equelle syntax that lets the user create functions with the same flexibility.

2.8 Mutable variables

Since variables are immutable by default, they must explicitly be declared mutable. This is done by writing `Mutable` in front of the type. Example usage:

```
regular : Scalar = 8
writable : Mutable Scalar = 9
writable = regular  # Ok
writable = 4        # Ok
regular = writable  # Error
regular = 6         # Error
```

3 Built-in operators and functions

3.1 Arithmetic operators

The four fundamental operators (`+ - * /`) are available. They can be used with `Scalar` or `Vector` expressions, with the exceptions that two `Vector` expressions cannot be multiplied or divided, and that a `Scalar` and a `Vector` expression cannot be added or subtracted.

If one expression is a `Collection`, then both expressions must be collections that are `On` the same domain. There is one exception to this: you can always multiply or divide a `Collection` with a `Scalar`.

The norm operator is written using vertical bars ($|\cdot|$) and can be used for multiple purposes. For a `Scalar` it returns the absolute value, for a `Vector` it returns the Euclidean norm, and for grid entities it returns the measure (volume, area, length) of the entity. It also works for collections, returning a `Collection Of Scalar` in all cases. An example:

```
ifaces = InteriorFaces()
first = FirstCell(ifaces)
second = SecondCell(ifaces)
x = |ifaces| * |Centroid(second) - Centroid(first)|
```

In the above, `x` will end up being a `Collection Of Scalar On InteriorFaces()`, and its value will be, for each interior face, the product of the face's area with the distance between its adjacent cells' centroids.

3.2 Comparisons and logical operators

The equality comparison operators `==` and `!=` are accepted by Equelle for all basic types. The numerical comparisons `<`, `<=`, `>` and `>=` are accepted

for `Scalar` types only. They all return `Bool`, or `Collection Of Bool` if the operands are collections. As for arithmetic operators, both collections must then be `On` the same domain.

The logical operators `And`, `Or`, `Not` and `Xor` can be used on either two `Bool` operands or two `Collection Of Bool` operands.

The trinary operator (`?  :`) takes three arguments. The first (before the question mark) must be a `Bool`, and the second and third arguments (on either side of the colon) can be of any type, but both must be of the same type. If the first argument is true, the second argument is returned, otherwise the third. If any argument is a collection, they must all be collections on the same domain, and the operator is applied element-wise.

3.3 Grid topology

The built-in functions dealing with grid topology are summarized in Table 2. Most of these functions simply return the canonical built-in domains of Equelle, they all return domains of the indicated type and are named like this:

`All|Interior|Boundary + Cells|Faces|Edges|Vertices`.

The functions `FirstCell` and `SecondCell` do not require domains for their arguments, nor do they return domains. In a grid, each face is oriented. This orientation is so that the `Normal` of a face points from its `FirstCell` to its `SecondCell`. These function may return `Empty` for faces on the outer boundary, in fact the boundary can be characterized by the fact that either `FirstCell` or `SecondCell` must be `Empty`.

The `IsEmpty` function returns `True` if the input is `Empty` and `False` otherwise.

Consider the following example:

```
bfaces = BoundaryFaces()
first = FirstCell(bfaces)
second = SecondCell(bfaces)
bface_cells = IsEmpty(first) ? second : first
```

The type of `bface_cells` will be `Collection Of Cell On BoundaryFaces()`. Note that some boundary faces can be mapped to the same cell. In the grid

Function	*Arguments*	*Returns*
`AllCells`		`Collection Of Cell`
`InteriorCells`		`Collection Of Cell`
`BoundaryCells`		`Collection Of Cell`
`AllFaces`		`Collection Of Face`
`InteriorFaces`		`Collection Of Face`
`BoundaryFaces`		`Collection Of Face`
`AllEdges`		`Collection Of Edge`
`InteriorEdges`		`Collection Of Edge`
`BoundaryEdges`		`Collection Of Edge`
`AllVertices`		`Collection Of Vertex`
`InteriorVertices`		`Collection Of Vertex`
`BoundaryVertices`		`Collection Of Vertex`
`FirstCell`	`Collection Of Face`	`Collection Of Cell`
`SecondCell`	`Collection Of Face`	`Collection Of Cell`
`IsEmpty`	`Collection Of Cell`	`Collection Of Bool`

Table 2: Grid topology built-in functions

Function	*Arguments*	*Returns*
`Centroid`	`Collection Of Cell`	`Collection Of Vector`
`Centroid`	`Collection Of Face`	`Collection Of Vector`
`Normal`	`Collection Of Face`	`Collection Of Vector`

Table 3: Grid geometry built-in functions

in Figure 1 for example, `bf_cells` would contain cell 6 twice, once for face i and once for face u. Therefore `bf_cells` cannot be a domain.

3.4 Grid geometry

The built-in functions dealing with grid geometry are summarized in Table 3.

All grid geometry functions return vectors. Note that `Normal` returns unit normals.

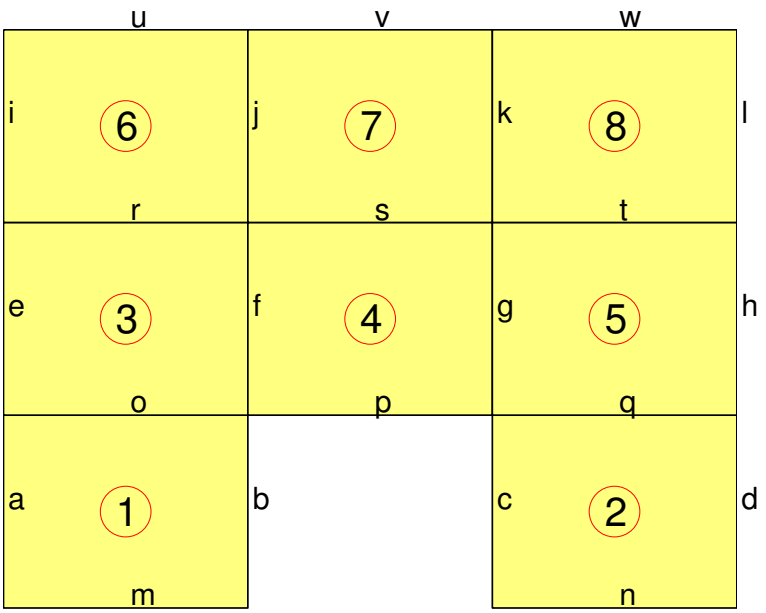

Figure 1: An example grid in 2D

3.5 On and Extend as operators

The operators `On` and `Extend` both create new collections from existing ones. We deal with `Extend` first.

`Extend` can be used in two ways. The first and simplest is to create a `Collection` from a single (non-collection) expression of a basic type, each element of the resulting collection being equal to the given single expression. For example we can extend a previous example:

```
bfaces = BoundaryFaces()
first = FirstCell(bfaces)
second = SecondCell(bfaces)
bface_cells = IsEmpty(first) ? second : first
bface_sign = IsEmpty(first) ? (-1 Extend bfaces) : (1 Extend bfaces)
```

In the above example, `(-1 Extend bfaces)` is a `Collection Of Scalar On bfaces` with each element equal to -1. The right hand side expression of the `Extend` operator must be a domain.

The second way to use `Extend` is to take a `Collection Of Scalar` and

extend it with zeros to form a new `Collection Of Scalar` on a larger domain. For example:

```
a = AllFaces()
b = BoundaryFaces()
c = Centroid(b)[0]  # The x coordinate of the face centroid
d = x Extend a
```

A collection can be thought of as a 1-1 mapping from its domain to its data elements. In the above example, c can be considered to take $f \in b \to c(f)$. Then d takes $f \in b \to c(f)$ and $f \in a \setminus b \to 0$. For this use of `Extend`, the expression on the right of the operator must be a domain that contains the domain of the collection of the left.

The `On` operator performs the evaluate-on or restrict-to operation on a collection. Given a collection, one can use `On` to form a new collection that is restricted to a subset of the domain of the original collection. For example:

```
all_centroids = Centroid(AllFaces())
boundary_centroids : Collection Of Vector On BoundaryFaces()
boundary_centroids = all_centroids On BoundaryFaces()
```

In the above, `boundary_centroids` would be identical to the result of `Centroid(BoundaryFaces())`. Note that when unlike `Extend`, the `On` operator can be used with other basic types than `Scalar`, since it does not need to assign a default value (zero for `Extend`) to any elements, but draws all its data from the collection given on the left hand side. The new collection's domain will be the same as that of the right hand side.

The right hand side does not need to be a domain in itself. It is however required to be a subset of the domain of the left hand side collection. Extending a previous example shows how:

```
bfaces = BoundaryFaces()
first = FirstCell(bfaces)
second = SecondCell(bfaces)
bface_cells = IsEmpty(first) ? second : first
bface_cell_centroids = Centroid(AllCells()) On bface_cells
```

In the above, the type of `bface_cells` is `Collection Of Cell On BoundaryFaces() Subset Of AllCells()`. It contains, for each boundary face, the cell just on the inside of the face. It is not a domain, since any cell can occur multiple times in the collection. The collection `bface_cell_centroids` contains, for each boundary face, the centroid of the cell just inside of it.

Another example:

```
a = AllCells()
c = Centroid(a)
s = SecondCell(InteriorFaces())
d = c On s
```

Viewing collections as 1-1 mappings from domains to data, one can say that if c takes an element $e \in a \rightarrow c(e)$ and $s \subset a$, then d (which is given by `c On s`) simply takes $e \in s \rightarrow c(e)$.

3.6 Discrete operators

The discrete operators `Gradient` and `Divergence` make it easy to program finite volume methods. Their properties are summarized in Table 4.

Function	*Arguments*	*Returns*
`Gradient`	`Collection Of Scalar On AllCells()`	`Collection Of Scalar On InteriorFaces()`
`Divergence`	`Collection Of Scalar On InteriorFaces()`	`Collection Of Scalar On AllCells()`
`Divergence`	`Collection Of Scalar On AllFaces()`	`Collection Of Scalar On AllCells()`

Table 4: Discrete operators

The function call `Gradient(x)` computes for each interior face, the difference between `x` at the two adjacent cells. It is equivalent to the following Equelle function:

```
grad : Function(x : Collection Of Scalar On AllCells()) ...
                -> Collection Of Scalar On InteriorFaces()
grad(x) = {
    first = FirstCell(InteriorFaces())
    second = SecondCell(InteriorFaces())
    -> (x On second) - (x On first)
}
```

The function call `Divergence(x)` computes for each cell, the discrete divergence of the flux `x`. This is equal to the orientation-signed sum of the elements of `x` belonging to faces adjacent to the cell. This sum is done with signs, in such a way that elements of faces that are positively oriented

with respect to the cell are added, and elements with negative orientation subtracted. Consider for example the grid in Figure 1. Assume that all faces are oriented so that the normals point to the right or upwards. Then the (signed) boundary of cell 3 is equal to $-e + f - o + r$. If those faces have associated fluxes equal to $\{0, 1, 2, -1\}$ the divergence at cell 3 would be $-0 + 1 - 2 + (-1) = -2$.

Note that there are two variants of `Divergence`: for convenience there is one that is defined on `InteriorFaces()` only, it is equivalent to the following function invoking the full `Divergence` function:

```
div_int : Function(x : Collection Of Scalar On InteriorFaces()) ...
                    -> Collection Of Scalar On AllCells()
div_int(x) = {
    -> Divergence(x Extend AllFaces())
}
```

It is not possible to define an Equelle function that is equivalent to the full `Divergence` built-in function, since there is no way (for now at least) to access the neighbour faces of a cell.

3.7 Solver functions

Equelle is not intended to write advanced linear solvers or similar low-level code. Instead, such features are meant to be provided by the back-end, through solver functions of the language. So far, there are only 2 such functions available, for solving nonlinear (or linear) problems with the Newton-Raphson method.

The simplest, `NewtonSolve`, solves a scalar equation. It takes as arguments a function and an initial guess, and returns a value that makes the function zero within some tolerance. What that tolerance is and how it is controlled is currently up to the back-end to specify. Since Equelle does not have a system for exceptions or other error handling, it is back-end dependent what happens if the algorithm fails to converge. The function argument, its return value, and the initial guess must all be of the same type, scalar collections over the same domain. A short example:

```
pressureResLocal(pressure) = {
    -> computePressureResidual(pressure, total_mobility, source)
}
```

```
p = NewtonSolve(pressureResLocal, p0)
```

In the above, there exists some function `computePressureResidual` taking three arguments. At the point shown we want to solve for the first argument, with fixed values for the two other arguments. We then create a local function binding the two last arguments, and pass the resulting function to `NewtonSolve` together with an initial guess.

For systems of equations `NewtonSolveSystem` is available. Its interface is very similar to the simple version. It takes as argument an `Array Of` functions and an `Array Of` initial guesses, and returns an `Array Of` solutions:

```
pressureResLocal : Function(pressure : Collection Of Scalar On AllCells(), ...
                            sw : Collection Of Scalar On AllCells()) ...
                            -> Collection Of Scalar On AllCells()
pressureResLocal(pressure, sw) = {
    total_mobility = computeWaterMob(sw) + computeOilMob(sw)
    -> computePressureResidual(pressure, total_mobility, source)
}

newvals = NewtonSolveSystem([pressureResLocal, transportResLocal], ...
                            [p0, 0.5 Extend AllCells()])
```

In the above partial example, we want to solve for `pressure` and `sw` simultaneously. Then we have to pass arrays to `NewtonSolveSystem`, and each of those functions needs to take both of the unknowns as inputs, as shown above for the `pressureResLocal` function. The `transportResLocal` function is not shown here, but that one also must take both `pressure` and `sw`.

3.8 Input and output

Equelle provides function to get user input, that are summarized in Table 5. All functions take a `String` argument, a tag for the operation. It is up to the back-end how that is used, for example the serial back-end writes a series of numbered files prefixed with the tag each time `Output` is called with the same tag. The function `InputCollectionOfScalar` takes a domain as input, and returns a collection over the same domain. The function `InputDomainSubsetOf` takes a domain as input argument, and returns a new domain that is a subset of the input domain. Ensuring that any user input actually satisfies this is the duty of the back-end. This function is notable

Function	*Arguments*	*Returns*
`InputScalarWithDefault`	`String, Scalar`	`Scalar`
`InputCollectionOfScalar`	`String`, *domain*	`Collection Of Scalar`
`InputDomainSubsetOf`	`String`, *domain*	*domain*
`InputSequenceOfScalar`	`String`	`Sequence Of Scalar`
`Output`	`String, Scalar`	
`Output`	`String, Collection Of Scalar`	

Table 5: I/O functions

Function	*Arguments*	*Returns*
`Dot`	2 `Collection Of Vectors`	`Collection Of Scalar`
`Sqrt`	`Collection Of Scalar`	`Collection Of Scalar`
`MinReduce`	`Collection Of Scalar`	`Scalar`
`MaxReduce`	`Collection Of Scalar`	`Scalar`
`SumReduce`	`Collection Of Scalar`	`Scalar`
`ProdReduce`	`Collection Of Scalar`	`Scalar`

Table 6: Miscellaneous functions

in particular, since it is the only way to create a new domain that is not equal to one of the 12 standard domains (`AllCells(), InteriorCells()` etc.).

3.9 Miscellaneous functions

Other built-in functions include the dot product, `Dot`, and the square root function, `Sqrt`. Also there are reduction functions that reduce a collection to its minumum, maximum, sum or product. See Table 6 for a summary.

4 Control structures and user-defined functions

4.1 Scopes

All variables can only be used in the scope they are declared, or scopes nested inside that. Scopes are delimited by curly braces ({*cdot*}) and currently occur as `For` loops or function definitions. Scopes can be arbitrarily nested, so a function can be declared inside a for loop inside a function and so on.

4.2 The `For` loop

Explicit loops are much more rare in Equelle than in many other languages. This is because operations on collections are the norm, reducing the need for such loops. The only use of `For` currently allowed is looping over a `Sequence`. That uses the following syntax:

```
# Type of timesteps : Sequence Of Scalar
For dt In timesteps {
    # Code that uses dt goes here.
}
```

Inside the loop, the loop variable (`dt` above) takes on the values of the sequence given (`timesteps` above) one at a time, in order. It is explicitly sequential and will not be parallelized. Inside the loop scope, the loop variable is immutable, although it takes on a new value every iteration.

4.3 Functions

Equelle supports defining functions in arbitrary scopes. Currently, the type of a function must be declared, as discussed in 2.7. After declaration, it may be defined. A short example:

```
# Declaring the type of upwind is necessary for now
upwind : Function(flux : Collection Of Scalar On InteriorFaces(), ...
                  x    : Collection Of Scalar On AllCells()) ...
                  -> Collection Of Scalar On InteriorFaces()
# Defining the function.
upwind(flux, x) = {
    x1 = x On FirstCell(InteriorFaces())
    x2 = x On SecondCell(InteriorFaces())
    -> flux >= 0.0 ? x1 : x2
}
```

The above shows the syntax for both declaring and defining a function. The function must end with a return statement, consisting of the return operator `->` and an expression.

A function can use any variables that are in the scope in which they are defined. The above example could instead have been written:

```
first = FirstCell(InteriorFaces())
```

```
second = SecondCell(InteriorFaces())
# Declaring the type of upwind is necessary for now
upwind : Function(flux : Collection Of Scalar On InteriorFaces(), ...
                  x    : Collection Of Scalar On AllCells()) ...
                  -> Collection Of Scalar On InteriorFaces()
# Defining the function.
upwind(flux, x) = {
    x1 = x On first
    x2 = x On second
    -> flux >= 0.0 ? x1 : x2
}
```

Notice that since variable in Equelle are immutable, their values cannot change between the point the function is defined and the calls to the function. The single, rare, exception, is that explicitly declared `Mutable` variables can be changed. In Equelle, a function does not store the state at its point of definition in any way, and if it uses a `Mutable` variable from its surrounding scope, its current value is used whenever the function is called.

5 The grammar of Equelle

The Equelle compiler is written in C++, and uses the compiler generator tools `flex` and `bison` for lexical scanning and parsing. The files `equelle_lexer.l` and `equelle_parser.y` contain the definitions of the scanner and parser, respectively.

5.1 Allowed identifiers

In Equelle, the lower-case letters `[a-z]`, upper-case letters `[A-Z]`, digits `[0-9]` and underscore `[_]` are allowed in identifiers. Built-in keywords and the names of built-in functions all start with an upper-case letter. User-defined identifiers can be used for the names of functions and variables, and must always start with a lower-case letter.

5.2 Syntax for literals

There are literal expressions available for the following types: `Scalar`, `Bool`, `String` and `Array`.

Any number in an expression will be interpreted as a floating-point number and inferred to be a `Scalar`. For the time being, Equelle does not support scientific notation for scalars however, such as `2.3e-6`. This applies only to numbers in the Equelle program itself, for example when reading input from files the back-end supports any allowable C++ notation, as readable by standard input stream facilities.

The literals for `Bool` are `True` and `False`.

A string literal consists of a sequence of characters surrounded by double quotes, and it is possible to use backslash as escape character, for example

```
example = "A string with \"escaped double quotes\"."
```