*Bull. Nov. Comp. Center*, Comp. Science, 24 (2006), 119–128


# Sisal 3.1 language structures decomposition*

A. P. Stasenko

**Abstract.** The paper describes equivalent transformations of Sisal 3.1 language structures in detail. The programming language Sisal 3.1 is based on Sisal 90. Transformations are aimed to to decomposition of complex language structures into more simple ones that can be directly expressed by an internal representation IR1 based on an intermediate form language IF1. Currently some description of similar transformations can be found in few works about Sisal 90 in the form of examples. The specified transformations are actually performed inside the front-end compiler from Sisal 3.1 into the internal representation IR1 and can be used to better understand its translation strategy. The paper also briefly (but sufficiently for understanding) describes the difference between Sisal 3.1 and Sisal 90.

## 1. Introduction

The Sisal 3.1 language [1], developed in the A.P. Ershov Institute of Informatics Systems, is a direct successor of Sisal 90 (Stream and Iteration in a Single Assignment Language) data-flow language [2] that was originally designed by collaborating teams from the Lawrence Livermore National Laboratory, Colorado State University, the University of Manchester and Digital Equipment Corporation. Sisal 3.1 simplifies, improves, extends and more exactly specifies Sisal 90. Sisal 3.1 also incorporates ideas of enhanced module support and preprocessing of Sisal 3.0 [3]. The most significant extension of Sisal 3.1 are function overloading and user-defined types, for which it is possible to define user-defined operations.

During translation, some complex Sisal 3.1 language structures need to be reduced to more unified objects of the internal representation IR1 [4] based on the intermediate form language IF1 [5]. Peculiarities of such transformations are shown in terms of Sisal 3.1 by decomposition of complex language structures into more simple ones that can be directly expressed by IR1. The aspects of parallelization are not covered in this paper, since the IF1 language is based on the data flow which is parallelized by the compiler back-end or run-time support system.

*Work was financially supported by the Ministry of Education of the Russian Federation (the scientific program "Universities of Russia", grants N UR.04.01.027 and N UR.04.01.201).

## 2. Decomposition of the case expression

The conditional expression **case** is naturally decomposed into the conditional expression **if** with additional **elseif** branches. Since the **if** expression with an arbitrary number of **elseif** branches can be already explicitly expressed by the IF1 language, its further simplification is not performed by the Sisal 3.1 front-end compiler and is not shown in this paper.

Every selection list of the **case** expression is transformed into one **if** or **elseif** condition using logical disjunction and conjunction operations over the comparison operation results: equality (=), "less than or equal to" ($\leq$) and "greater than or equal to" ($\geq$). For expressions "**case tag**" and "**case type**", the infix operation **tag** (**tag** function of Sisal 90) and the expression "**type** [...]" are used, respectively.

## 3. Decomposition of the multidimensional loops

Let us consider the following $n$-ary $m$-dimensional loop, where each reduction corresponds to one loop dimension (for the simplicity of further notation):

**for** $D_1$ **cross** $D_2$ **repeat**
  $B$
  **returns** $RN_1$ **of** $RV_1$;
  ...;
  $RN_n$ **of** $RV_n$
**end for**

The name $D_1$ denotes the loop range generator part without the operator **cross** and multidimensional indices of the **at** construction, the name $D_2$ denotes the remaining part of the range generator, the name $RN_{i\in 1\ldots n}$ denotes the reduction name with possible initial values and the name $RV_i$ denotes reduction loop values. In that notation $m$-dimensional loop expression can be decomposed into the following two loop expressions of dimensions 1 and $m-1$:

**for** $D_1$ **repeat**
  $\overline{x_1}$, ..., $\overline{x_n}$ := **for** $D_2$ **repeat**
    $B$
    **returns** $RN'_1$ **of** $RV_1$;
    ...;
    $RN'_n$ **of** $RV_n$
  **end for**
  **returns** $RN''_1$ **of** $\overline{x_1}$;
  ...;
  $RN''_n$ **of** $\overline{x_n}$
**end for**

The name $\overline{x_i}$ here and any other name with overline without a special note later denote any unique name. The names $RN_i'$ and $RN_i''$ depend on the name $RN_i$ as shown in Table 1.

**Table 1.** Decomposition rules for multi-dimensional reductions, that show how to determine the names $RN_i'$ and $RN_i''$, which are used in this section before, from the name $RN_i$

| Value of the $RN_i$ name | $RN_i'$ | $RN_i''$ |
|---|---|---|
| Equals to value, product, least, greatest, catenate, "catenate (...)" or user-defined reduction. | $RN_i$ | value |
| Equals to "**array** $[k]$[1]$(i_1, \ldots, i_k)$", where:<br>• part "$[k]$" is optional and equals to "$[m]$" by default;<br>• last indices of the part "$(i_1, \ldots, i_k)$" are optional like this whole part and equal to 1 if omitted. | $k > 1$ **array** $[k-1]$ $(i_2, \ldots, i_k)$<br>$k = 1$ **array** [1] $(i_2, \ldots, i_k)$ | **array** $(i_1)$<br>catenate $(i_1)$ |
| Equals to "**stream** $[k]$[1]", where part "$[k]$" is optional and equals to "$[m]$" by default. | $k > 1$ **stream** $[k-1]$<br>$k = 1$ **stream** [1] | **stream**<br>catenate |

If the range generator contains multidimensional indices "$n$ **in** $S$ **at** $j_1$, ..." before the operator **cross**, then the multidimensional loop can be represented in the following way:

**for** $D_3$ $n$ **in** $S$ **at** $j_1$ , $D_4$ **repeat**
  $B$
  **returns** $RN_1$ **of** $RV_1$ ;
  ...;
  $RN_n$ **of** $RV_n$
**end for**

The name $D_3$ denotes the range generator part without the operator **cross** and multidimensional indices of the construction **at**, the name $S$ denotes the array or stream source of multidimensional indices, the name $D_4$ denotes the remaining part of the range generator. In that notation $m$-dimensional loop expression can also be decomposed into the following two loop expressions of dimensions 1 and $m-1$ (where the names $RN_i'$ and $RN_i''$ depend on the name $RN_i$ as shown in Table 1):

[1]In Sisal 3.1 **array** $[k]$ and **stream** $[k]$ notation replaces array_$k$d and stream_$k$d notation of Sisal 90.

```
for D3 n1 in S at j1 repeat
  x1, ..., xn := for n in n1 at D4 repeat
                   B
                   returns RN'1 of RV1;
                           ...;
                           RN'n of RVn
                 end for
  returns RN''1 of x1;
          ...;
          RN''n of xn
end for
```

## 4. Decomposition of the array element selection

Let us represent the element selection expression from the array $A$ as "$A$[ *selection construction* ]". Its notation in Sisal 3.1 differs from that in Sisal 90 only by using the symbol "!" instead of ":" to separate parts of the triplets (the meaning of ":" in Sisal 3.1 was changed to more clearly represent the postfix typecast operation).

If a selection construction does not have the **cross** (or comma) operator, then it can be represented as "$D_1$ **dot** $D_2$ **dot** ... **dot** $D_m$", where $m \geq 1$ and all expressions $D_1$, ..., $D_m$ are ranges (as required by the operator **dot** semantics). If $m = 1$ and the part $D_1$ is a singlet, then the array element selection operation can be represented directly in the IR1 and does not require further decomposition, otherwise the array element selection operation can be decomposed into the following one-dimensional loop:

```
for x1 in D1 dot x2 in D2 dot ... xm in Dm
  A1 := A [ x1, x2, ..., xm ]
  returns array of A1
end for
```

The name $x_i$ (here and later) denotes any unique name, if the part $D_i$ does not have the form "*name* $N$ **in** $D_i$", and denotes the name $N$ otherwise. If the selection construction contains the operator **cross** (or comma), then it can be represented as "$S_1$, $S_2$, ..., $S_m$ **cross** $C_1$" or "$D_1$ **dot** $D_2$ **dot** ... **dot** $D_m$ **cross** $C_2$", where $S_1$, ..., $S_m$ denote singlets and the names $C_1$ and $C_2$ denote the remaining parts of the selection construction and additionally the part $C_1$ does not begin with a singlet.

The array element selection operation beginning with a singlet can be decomposed into the following **let** expression (further decompositions need to be applied recursively):

```
let A1 := A [ S1 , S2 , ... , Sm ]
  in A1 [ C1 ]
end let
```

The array element selection operation beginning with a range can be decomposed into the following one-dimensional loop (further decompositions need to be applied recursively):

```
for x1 in D1 dot x2 in D2 dot ... xm in Dm repeat
  A1 := A [ x1 , x2 , ... , xm ]
  returns array of A1 [ C2 ]
end for
```

The presented decomposition of the array element selection operation also explains an additional restriction, which is missed in Sisal 90 user manual, for the selection construction triplets with omitted parts: they must be placed as the first operand of the selection construction or just after the **cross** operator. In the range $D_1$, the first and second omitted triplet parts are explicitly represented via "liml ($A$)" and "limh ($A$)", correspondingly. In the ranges $D_2, \ldots, D_m$ the triplet parts cannot be omitted because there is no corresponding univocal array dimension available whose lower and upper bounds can be taken. In summary, an arbitrary array element selection operation was decomposed into the array element selection with simple indices.

## 5. Decomposition of the array element replacement

In this section, we continue to use the notation of selection construction introduced before. The array element replacement expression in a general form looks like "$A$ [ *selection construction* := *replacement construction* ]". The array element replacement operation in Sisal 3.1 differs from the same operation of Sisal 90 only by using the symbols ":=" with a more clear meaning instead of the symbol "!" between the selection and replacement constructions. It is further shown that any array element replacement expression can be decomposed into the array one-element replacement with simple index.

### 5.1. Array element replacement with a singlet list selection construction

If the selection construction is a singlet list $S_1, \ldots, S_n$, then the replacement construction is allowed to be an expression list $E_1, \ldots, E_t$ and the array element replacement operation is elementary represented as a composition of the following one-element replacement operations:

$A$ [ $S_1$ , ... , $S_n$ := $E_1$ ]
[ $S_1$ , ... , ( $S_n$ ) + 1 := $E_2$ ]
...
[ $S_1$ , ... , ( $S_n$ ) + (t −1) := $E_t$ ]

### 5.2. Array element replacement with a scalar replacement construction

Let us consider the case then the selection construction is not a singlet list and the replacement construction is an expression of type of the $n$-th dimension of the array $A$, where $n$ is the number of the selection construction ranges and singlets. In this case the array element replacement operation can be decomposed into nested one-dimensional loops, obtained after the recursive application of the decompositions given below.

If the selection construction does not have the **cross** (or comma) operator, the array element replacement operation can be decomposed into the following one-dimensional loop:

**for** $x_1$ **in** $D_1$ **dot** $x_2$ **in** $D_2$ **dot** ... $x_m$ **in** $D_m$
  $A$ := **old** $A$ [ $x_1$ , $x_2$ , ... , $x_m$ := *replacement construction* ]
  **returns** value **of** $A$
**end for**

The array element replacement operation beginning with a singlet can be decomposed into the following **let** expression (further decompositions need to be applied recursively):

**let** $\overline{A_1}$ := $A$ [ $S_1$ , $S_2$ , ... , $S_m$ ]
  **in** $\overline{A_1}$ [ $C_1$ := *replacement construction* ]
**end let**

The array element replacement operation beginning with a range can be decomposed into the following one-dimensional loop (further decompositions need to be applied recursively):

**let** $\overline{A_1}$ := A
  **in for** $x_1$ **in** $D_1$ **dot** $x_2$ **in** $D_2$ **dot** ... $x_m$ **in** $D_m$
    $\overline{A_2}$ := **old** $\overline{A_1}$ [ $x_1$ , $x_2$ , ... , $x_m$ ];
    $\overline{A_3}$ := $\overline{A_2}$ [ $C_2$ := *replacement construction* ];
    $\overline{A_1}$ := **old** $\overline{A_1}$ [ $x_1$ , $x_2$ , ... , $x_m$ := $\overline{A_3}$ ]
    **returns** value **of** $\overline{A_1}$
  **end for**
**end let**

### 5.3. Array element replacement with array replacement construction

Let us consider the case then the selection construction is not a singlet list and the replacement construction is an expression of type of a $k$-dimensional array of elements that have the type of the $n$-th dimension of the array $A$. In the considered case, $k$ should be a sum of ranges in the selection construction minus the number of its **dot** operators. In this case the array element replacement operation can be also decomposed into nested one-dimensional loops, obtained after the recursive application of decompositions given below.

If the selection construction does not have the **cross** (or comma) operator, the array element replacement operation can be decomposed into the following one-dimensional loop:

```
let ī := 1
  in for x1 in D1 dot x2 in D2 dot ... xm in Dm
       A := old A [
         x1 , x2 , ... , xm :=
         ( replacement construction ) [ ī ]
       ];
       ī := old ī + 1
       returns value of A
     end for
end let
```

The array element replacement operation beginning with a singlet can be decomposed into the following **let** expression (further decompositions need to be applied recursively):

```
let Ā1 := A [ S1 , S2 , ... , Sm ]
  in Ā1 [ C1 := replacement construction ]
end let
```

The array element replacement operation beginning with a range can be decomposed into the following one-dimensional loop (further decompositions need to be applied recursively):

```
let Ā1 := A; ī := 1
  in for x1 in D1 dot x2 in D2 dot ... xm in Dm
       Ā2 := old Ā1 [ x1 , x2 , ... , xm ];
       Ā3 := Ā2 [ C2 := ( replacement construction ) [ ī ] ];
       Ā1 := old Ā1 [ x1 , x2 , ... , xm := Ā3 ];
       ī := old ī + 1
       returns value of Ā1
```

```
    end for
end let
```

## 6. Decomposition of the "where" expression

The **where** expression in Sisal 3.1 was seriously reconsidered since Sisal 90. In Sisal 90, it has a limited case-based design, requiring all conditions to be functions of the same array, that is unnatural and needs a special handling by a parser. In Sisal 3.1, the **where** expression has the following form:

**where** *n-dimensional array A* **is** *name I* **in**
  *expression R*
**end where**

The new **where** expression is semantically more powerful because:

- it does not require the expression $R$ to be the case expression;
- it is easier to parse due to the isolation of the control array $A$ on the syntax level;
- it is more natural due to the semantics of the name $I$, representing an array $A$ element, in the expression $R$, that removes duality of the control array name in the Sisal 90 **where** expression.

The Sisal 3.1 **where** expression is decomposed into nested one-dimensional loops in the following way:

**for** $\overline{A_1}$ **in** $A$ **returns array of**
  **for** $\overline{A_2}$ **in** $\overline{A_1}$ **returns array of**
    ...
    **for** $I$ **in** $\overline{A_{n-1}}$ **returns array of**
      *expression R*
    **end for**
    ...
  **end for**
**end for**

## 7. Decomposition of the vector operations

In addition to arithmetic, relational and boolean vector operations of Sisal 90, Sisal 3.1 allows any infix, prefix and postfix operations (including user defined operations) to be extended to their vector forms. Sisal 3.1 also allows vector operations between streams and arrays that produce streams. All vector operations are decomposed into one-dimensional loops. An operation on multidimensional vectors is decomposed into a vector operation on vectors of lower dimensions.

Prefix and postfix operations on arrays *op* $(A)$ are decomposed into:

```
for i in A
  returns array(liml(A)) of op (i)
end for
```

Prefix and postfix operations on streams *op* $(S)$ are decomposed into:

```
for i in S
  returns stream of op (i)
end for
```

An infix operation *op* on two arrays $A_1$ and $A_2$ is decomposed into:

```
for i1 in A1 dot i2 in A2
  returns array of i1 op i2
end for
```

An infix operation *op* on array $A$ and stream $S$ is decomposed into:

```
for ia in A dot is in S
  returns stream of ia op is
end for
```

An infix operation *op* on array $A$ and non-vector value $V$ is decomposed into:

```
for i in A
  returns array(liml(A)) of i op V
end for
```

An infix operation *op* on stream $S$ and non-vector value $V$ is decomposed into:

```
for i in S
  returns stream of i op V
end for
```

## References

[1] Stasenko A.P. Basic means of the Sisal 3.1 language. — Novosibirsk, 2006. — 60 p. — (Prepr. / Siberian Division of the Russian Academy of Sciences. A.P. Ershov Institute of Informatics Systems; N 132) (in Russian).

[2] Feo J.T. Sisal 90 user's guide / Feo J.T., Miller P.J., Skedzielewski S.K. and Denton S.M. — Livermore, CA: Lawrence Livermore National Laboratory, Draft 0.96, 1995. — 80 p.

[3] Kasyanov V.N., Biryukova Yu.V., Evstigneev V.A. A functional language Sisal 3.0 // Supercomputing support and Internet-oriented technologies. — Novosibirsk, 2001. — P. 54–67 (in Russian).

[4] Stasenko A.P. Internal representation of functional programming system Sisal 3.0. — Novosibirsk, 2004. — 54 p. — (Prepr. / Siberian Division of the Russian Academy of Sciences. A.P. Ershov Institute of Informatics Systems; N 110) (in Russian).

[5] Skedzielewski S.K., Glauert J. IF1 – An intermediate form for applicative languages, version 1.0. — Livermore, CA, 1985. — 68 p. — (Tech. Rep. / Lawrence Livermore National Laboratory; M-170).