**KELDYSH INSTITUTE OF APPLIED MATHEMATICS**
**Russian Academy of Sciences**

**Sergei M. Abramov, Sergei A. Romanenko**

# How to Use Vectors for Representing Ground Expressions in the Implementations of the Language Refal

**Moscow**
**1988**

**Sergei M. Abramov, Sergei A. Romanenko. How to Use Vectors for Representing Ground Expressions in the Implementations of the Language Refal**

Most implementations of the language Refal represent ground expressions by doubly-linked lists, which results in low-cost concatenation. On the other hand, that representation necessitates the real copying of expressions in all cases where a variable's value has to be used two or more times. The high cost of copying often prevents the user from writing Refal programs in truly functional style. The paper describes an alternative representation of ground expressions, which allows the copying of expressions to be reduced to copying of pointers to the expressions. The concatenation of expressions, however, becomes more expensive.

**Key Words and Phrases:** Refal, Lisp, symbol manipulation, garbage collection, compaction, storage management, pointer readjustment.

## Contents

## 1 The Problem of Copying

Everybody who has been taught how to write programs in Refal [Tur79, Tur86] is unlikely to forget the inarticulate fear and mystical trepidation inspired by the necessity to copy ground expressions. Most Refal text-books vividly describe the perils of copying, and suggest numerous saving remedies and means of escape [Tur71, Tur77, Kor86, Tur89].

However ingenious those remedies may seem, they do obscure the meaning of Refal programs, as the user is forced to write programs in imperative, rather than in functional, style.

As an example, let us define a Refal function SUBST that is to be called in the following way:

$$\texttt{<SUBST (}\mathcal{E}_{tab}\texttt{) }\mathcal{E}_e\texttt{>}$$

where $\mathcal{E}_e$ is a ground expression, and $\mathcal{E}_{tab}$ a "substitution table", which is a sequence of ordered pairs

$$(\mathcal{S}_1\mathcal{E}_1)\ \ (\mathcal{S}_2\mathcal{E}_2)\ \ \ldots\ \ (\mathcal{S}_n\mathcal{E}_n)$$

where $\mathcal{S}_i$ are some symbols, and $\mathcal{E}_i$ some ground expressions. For each symbol $\mathcal{S}$ appearing in the expression $\mathcal{E}_e$, the function SUBST must examine the table $\mathcal{E}_{tab}$ to see whether it contains a pair $\mathcal{S}_i\mathcal{E}_i$, such that $\mathcal{S} = \mathcal{S}_i$. If so, the occurrence of the symbol $\mathcal{S}$ in $\mathcal{E}_e$ must be replaced with $\mathcal{E}_i$. Otherwise, the occurrence of $\mathcal{S}$ must remain unchanged. For instance:

```
<SUBST ( (A X X X)(B Y Y Y) )
               A B C (A C B)() B>  -->
  X X X Y Y Y C (X X X C Y Y Y) () Y Y Y
```

A straightforward definition of SUBST—which can be written in Refal without any difficulty—is shown in Figure 1.

```
<SUBST tT > =
<SUBST tT sX eY> =
    <LOOKUP tT sX> <SUBST tT eY>
<SUBST tT (eX) eY> =
    ( <SUBST tT eX> ) <SUBST tT eY>
<LOOKUP (eA (sX eV) eB) sX>  =  eV
<LOOKUP (eT) sX>  =  sX
```

Figure 1: A functional-style definition of SUBST.

This definition, however, would be severely criticized by the authors of standard Refal text-books for making no effort to avoid copying the substitution table. In accordance with the standard approach, the "naïve" definition ought to be "improved" by making use of clever programming tricks. For example, applying a famous trick known as "expression traversal" [Tur71, Tur77, Tur89], we get a new definition of SUBST shown in Figure 2, which traverses expressions by means of two stacks.

```
<SUBST tT eX> = <THRU tT (()) eX ()>
<THRU (eA (sX eV) eB)
      (eP (eQ)) sX eY (eR)> =
   <THRU (eA (sX eV) eB)
         (eP (eQ eV)) eY (eR)>
<THRU (eT) (eP (eQ)) sX eY (eR)> =
   <THRU (eT) (eP (eQ sX)) eY (eR)>
<THRU (eT) (eP) (eX) eY (eR)> =
   <THRU (eT) (eP ()) eX ((eY) eR)>
<THRU (eT) (eP (eQ) (eX)) ((eY) eR)> =
   <THRU (eT) (eP (eQ (eX))) eY (eR)>
<THRU (eT) ((eQ)) ()> = eQ
```

Figure 2: An imperative-style definition of SUBST.

It can be easily seen that the second definition of SUBST is, essentially, an imperative one: it is tail-recursive and generates neither nested nor parallel function calls. Thus, the computation may be considered as "control-driven", rather that "data-driven", which, in certain situations, is undesirable (for example, it may be a hindrance to parallel execution in multiprocessor systems). On the other hand, the first "naïve" definition of SUBST is quite functional, and its parallelism can be easily seen.

## 2 Alternative Representations of Ground Expressions

### 2.1 Doubly-Linked Lists

The strong dislike shown by Refal text-books for copying expressions seems to be rather strange in view of the fact that the freedom from constrains on the use of copying would enable the programmer to write more concise and natural programs. This dislike, however, becomes understandable, if we take into account the way in which ground expressions are kept in computer memory by most popular Refal implementations.

Namely, ground expressions are usually represented by doubly-linked lists, each cell of a list having the structure shown in Figure 3.

The fields prev and next are used for linking cells to form linear sequences.

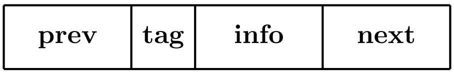


Figure 3: A cell of a doubly-linked list.

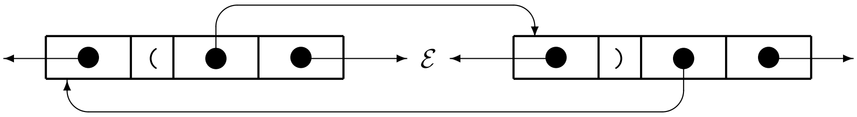


Figure 4: "Classical" representation of the term `(`$\mathcal{E}$`)`.

The field `prev` holds a pointer to the previous cell, and the field `next` to the next one.

The main advantage of doubly-linked lists is that the concatenation of two expressions amounts to modifying two pointers, the first one being the field `next` in the last cell of the first expression, and the second one the field `prev` in the first cell of the second expression. Consequently, the cost of concatenation does not depend on the size of the two expressions.

The field `tag` contains flags indicating the meaning of the information held in the field `info`.

The fields `tag` and `info` can be used in (at least) two different ways, resulting in two different representations of ground expressions. These representations will be—rather arbitrarily—referred to as "classical" and "conciliatory" ones.

In the case of the *"classical"* representation, each cell corresponds either to a symbol or to a parenthesis. If a cell represents a left parenthesis, the field `info` contains a pointer to the cell representing the corresponding right parenthesis, and *vice versa* (see Figures 4 and 5). The lists organized in this way were used as early as in the middle 60s in the implementation of the system AMBIT [Chr66].

It is obvious that, in the case of the "classical" representation, the cost of copying an expression is directly proportional to the expression's size.

Most Refal implementations are based on the "classical" representation [FOT69, KRT72, Tur77, Rom87a].

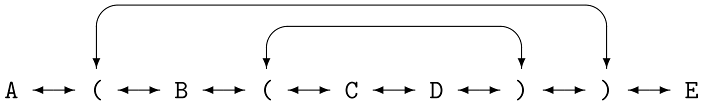


Figure 5: "Classical" representation of the expression `A (B (C D)) E`.

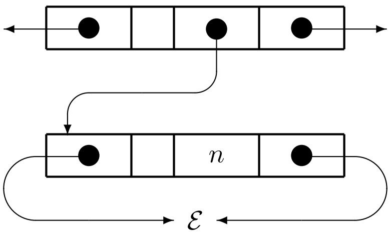


Figure 6: "Conciliatory" representation of the term `(`$\mathcal{E}$`)`.

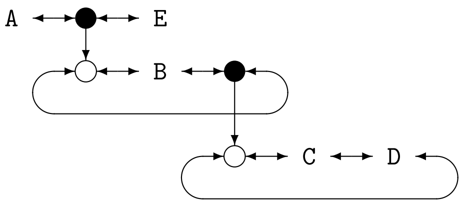


Figure 7: "Conciliatory" representation of the expression `A (B (C D)) E`.

In the case of the *"conciliatory"* representation, each cell corresponds to a Refal term, which is either a symbol or a pair of parentheses enclosing an expression. If a cell represents a pair of parentheses, the field `info` contains a pointer to another cell, called the "head" of the contents of the parentheses. The beginning and the end of the contents are linked to the head, thereby forming a circular structure (see Figures 6 and 7).

If two or more cells represent parentheses enclosing identical expressions, they may share a common head. Therefore, the copying of an expression can be reduced to copying the cells representing the top level of the expression, whereas the copying of the subexpressions enclosed in parentheses can be reduced to copying pointers to the heads of the subexpressions. In each head the field `info` can be used for counting references to the head. The lists organized in this way were used as long ago as in the early 60s in the system SLIP [Wei63].

## 2.2 Vectors

The "conciliatory" representation of expressions gives only a partial solution to the problem of copying. First, the top level of the expressions still has to be copied. Second, copying the contents of a pair of parentheses can be avoided only in cases where the whole subexpression is to be copied (in which case the copying amounts to copying the pointer to the head of the subexpression). But, in the case of copying a proper part of a subexpression, the top level of the

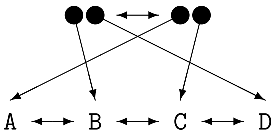


Figure 8: The expression `(B C D) (A B C)` represented with double references.

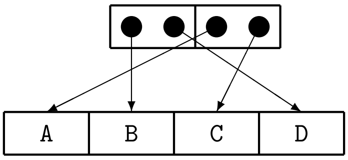


Figure 9: "Vector" representation of the expression `(B C D) (A B C)`.

subexpression still has to be copied.

As an example, consider the function `LR`:

```
<LR eX>          =    <LEFT eX> <RIGHT eX>
<LEFT  tA eP>  =    (eP)
<RIGHT eQ tB>  =    (eQ)
```

If we try to evaluate the following function call

```
<LR A B C D>   -->   (B C D) (A B C)
```

we will see that the fragments of the expression `A B C D` cannot be used for building the terms `(B C D)` and `(A B C)`, in view of the fact that the common subexpression `B C` cannot be included into two different linear sequences by modifying the contents of the fields `prev` and `next`.

We can, however, overcome the difficulty by changing the representation of parentheses. Namely, each pair of parentheses can be represented by a single cell containing two pointers: to the beginning and to the end of the subexpression enclosed in the parentheses (see Figure 8). This enables us to get rid of the head cells, and provides a means for referencing common subexpressions.

But now the following question arises: what benefit can be gained from the fields `prev` and `next`? They were previously used for reducing the expression concatenation to adjusting the contents of these fields. But now that we have permitted sharing of common subexpressions, the concatenation of expressions necessitates the copying of the top level of the operand expressions. Hence, there is no point in having these fields, and, eliminating them, we come to the "vector" representation of expressions (see Figures 9 and 10).

In the case of the *"vector"* representation, all the cells representing the top-level terms of an expression are arranged in computer memory so as to form a

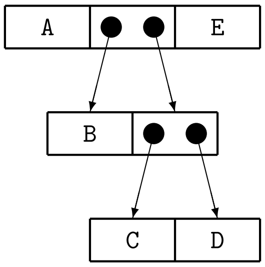


Figure 10: “Vector” representation of the expression `A (B (C D)) E`.

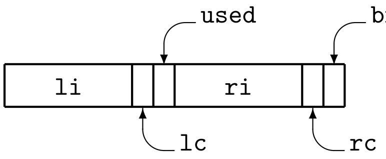


Figure 11: A cell of the “vector” representation.

one-dimensional array. Each cell has the configuration depicted in Figure 11, and includes the “left information” field `li`, the “right information” field `ri`, and the one-bit flag `br` determining the meaning of the fields `li` and `ri`.

If `br` $= 1$, the cell represents an expression enclosed in parentheses. If the contents of the parentheses is non-empty, `li` and `ri` contain pointers to the first cell and to the last cell of this expression, respectively. If the contents of the parentheses is empty, `li` $=$ `ri` $= 0$.

If `br` $= 0$, the cell represents a symbol. In this case `li` and `ri` contain some information about the symbol, which will be of little interest to us.

Besides, each cell contains three more one-bit fields: `used`, `lc`, and `rc`, whose purpose will be explained later.

If $\mathcal{E}$ is a ground expression, the number of symbols and parentheses appearing in $\mathcal{E}$ will be referred to as the *size* of the expression. If $\mathcal{E} = \mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_k$, where $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_k$ are ground terms, the number $k$ will be referred to as the *length* of the expression $\mathcal{E}$. For instance, the expression `A (B C) ()` has the size 7 and the length 3.

As can be easily seen, the “vector” representation allows the copying of an expression to be reduced to the copying of a pair of pointers, the cost of copying being unaffected by the size of the expression.

On the other hand, the concatenation of two expressions necessitates the creation of a new vector, whose length is the sum of the lengths of the operand expressions. Therefore, the cost of concatenation is directly proportional to the sum of the lengths of the operands.

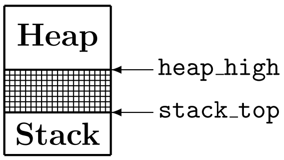


Figure 12: Dynamic memory allocation.

## 2.3 Memory Allocation

In the case of the "vector" representation, we have to use a storage management system enabling vectors of arbitrary size to be allocated on request. Thus it seems reasonable to adopt the following memory allocation scheme.

We assume that the storage space from which vectors are dynamically allocated is a contiguous area of the memory (see Figure 12). One part of this area, referred to as the *heap*, is used for storing ground expressions, whereas the rest of the area contains the *stack* (if one is needed).

In the following, the heap is assumed to reside at the beginning of the area (i.e. at low addresses), and the stack at its end (i.e. at high addresses). However, we could have placed—equally well—the heap at the high addresses, and the stack at the low addresses.

The variable `heap_high` contains a pointer to the last cell of the heap, and the variable `stack_top` to the top element of the stack.

The area between the heap and the stack contains free cells. So, if a contiguous group of cells is requested, the storage manager occupies the cells lying just after the heap, and, accordingly, increases the value of `heap_high`.

# 3 Garbage Collection

## 3.1 What is the Use of Garbage Collection?

Since we have allowed arbitrary subexpressions to be shared by different expressions, we need a means for identifying and reclaiming unused cells. This process is usually referred to as *garbage collection* [Hen80].

Moreover, the storage manager may run into the situation where, despite there being plenty of free cells, it is unable to find a sufficiently large block of adjacent unused cells. Therefore, the garbage collector must be able to perform *compaction* of all used cells into one end of the memory, so that the other end will contain a single block of adjacent unused cells.

The one-bit fields `used`, `lc`, and `rc` mentioned above are used during garbage collection, and, otherwise, must be equal to zero.

The field `used` is used for marking cells accessible to the program. The purpose of the fields `lc` (left cluster) and `rc` (right cluster) will be explained later.

## 3.2 An Informal Description of the Algorithm

The garbage collection algorithm comprises two separate phases: *marking* and *compaction.*

The task of the *marking* phase is to find all cells that can be accessed by the user's program, and to mark them by setting their fields `used` to 1. The algorithm takes as input a set of initial pointers to "directly accessible" cells, and then marks all cells that can be accessed through that set (directly or indirectly).

The task of the *compaction* phase is to move the marked cells to their new locations, and to update all pointers to these cells contained in other marked cells. Compaction is the most complicated phase of the garbage collection algorithm.

The main difficulty is that there is no limitation on the number of references to a cell (contained in other cells), and all such references have to be updated.

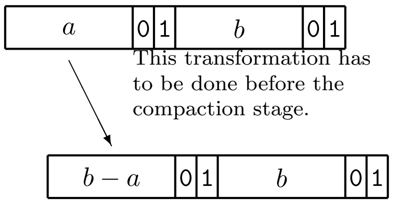
In the following, for brevity's sake, all marked cells containing pointers to a cell will be referred to as "masters" of that cell, and that cell will be called their "slave".

Suppose we are relocating a cell, and want to readjust all the references contained in its masters and pointing to the cell. This could be done, if we knew the cell's new address, the cell's old address, and the addresses of all masters. It is clear that the main difficulty is to find all the masters of the cell. As suggested by Morris [Mor78, Coh81, CN83], this problem can be solved by linking all the masters of each cell into a linear list (which will be referred to as the "cluster" of the cell's masters). Then, when relocating a cell, we shall be able to find all the masters and to readjust the pointers to the cell contained in the masters.

Unfortunately, a straightforward attempt at implementing this plan runs into the following contradiction. On the one hand, the masters of a cell must eventually be relocated. On the other hand, relocating any of the cell's masters is sure to corrupt the linear list of the masters.

This obstacle can be overcome in the following way. We can start by constructing the clusters for all cells. Then we can readjust all pointers without relocating the cells, thereby eliminating all the clusters before the actual relocation of the cells. (When readjusting the pointers to a cell, the only thing we need to know is the address of its new location.) Thus we get the state in which all cells are at their old locations, but all pointers have been readjusted to the new locations of the cells. Therefore, now the cells can safely be moved to their new locations.

One more difficulty still remains: each cell such that `br` $= 1$ contains *two* pointers, and both of them must be readjusted! Therefore, that cell may be the master of two different cells, in which case it has to be included in two different clusters. This problem can be solved by means of a simple trick. As can be



$a$
0
1
$b$
0
1
This transformation has to be done before the compaction stage.
$b-a$
0
1
$b$
0
1

Figure 13: How a double reference is replaced with a single one.

easily seen, when an expression is moved to a new place, its length—which is equal to the number of its top-level terms—remains unchanged. Therefore, if a cell contains pointers to the beginning and the end of the expression in the fields `li` and `ri`, the difference of the pointers held in `ri` and `li` does not change when the expression is moved.

Thus, by replacing the contents of the field `li` with the difference of the pointers held in `li` and `ri` (for all cells representing a pair of parentheses), we can halve the number of pointers that have to be readjusted (see Figure 13).

This transformation will be referred to as "replacing a double reference with a single one", and the inverse transformation as "replacing a single reference with a double one".

Now the cluster of a cell's masters can be built in the following way. The field `li` will be used for referencing the first master, and the field `ri` for linking the masters into a linear list. The fields `lc` and `rc` will be used as follows. If `lc` $= 1$, it will mean that the cell's cluster is non-empty. If `rc` $= 1$, it will mean that the cell belongs to a cluster, and `ri` contains a pointer to the next cell in the cluster (see Figure 14).

It should be noted that the information that was initially held in the field `li` in the slave cell has to be saved somewhere else, as the contents of `li` must be restored after the cell has been relocated. So, this information is put into the field `ri` of the last cell in the linear list of masters. This can be done, because that field is needed for no other purpose.

Thus we come to the following six-pass garbage collection algorithm:

1. Marking.
2. Replacing double references with single ones.
3. Building clusters.
4. Readjusting pointers.
5. Relocating cells.
6. Replacing single references with double ones.

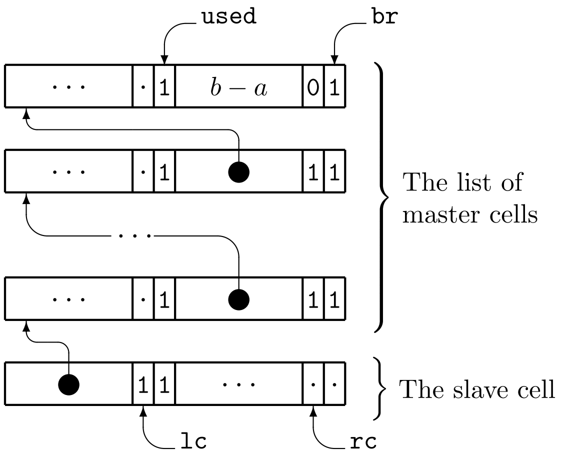


Figure 14: A slave cell and the cluster of its masters.

## 3.3 A Six-Pass Algorithm

Now we can give a detailed description of the six-pass garbage collection algorithm written in the language C [KR88].

The computer memory will be represented by a one-dimensional array of the size `MEMORY_SIZE`, each element of the array representing a cell. The size of a memory address is assumed to be equal to `ADDR_SIZE` bits, and the cells to be stored in the memory area between two addresses `MEMORY_LOW` and `MEMORY_HIGH`−1.

```
#define ADDR_SIZE 14
#define MEMORY_SIZE 1000

typedef unsigned int address;
typedef unsigned int bool;

/* A cell representing a term. */
struct term
  {
  address  li:   ADDR_SIZE;
  bool     lc:   1;
  bool     used: 1;
  address  ri:   ADDR_SIZE;
  bool     rc:   1;
  bool     br:   1;
  };
```

```
struct term t[MEMORY_SIZE];

#define MEMORY_LOW  1
#define MEMORY_HIGH 1000
```

Garbage collection is performed by the function `garb_coll()`, which, in turn, calls the function `mark_term()`.

The function `mark_term()` takes a pointer to a term and marks all terms accessible from that term.

To simplify the presentation, we assume that, when the garbage collection starts, there is only one immediately accessible cell: the one located at the address `MEMORY_LOW`. In the case of a real Refal implementation, of course, there may be several immediately accessible cells.

```
void mark_term(address m) {
  address k, k_max;
  if (!t[m].used) {
    t[m].used = 1;
    if (t[m].br && t[m].ri != 0) {
      k_max = t[m].ri;
      for (k=t[m].li; k<=k_max; k++)
        mark_term(k); }}
  return; }

void garb_coll(void) {
  address  m, k, new_place;

  /* The marking phase. */
  /* For the sake of simplicity, we  */
  /* assume that there is only one   */
  /* immediately accessible term.    */
  mark_term(MEMORY_LOW);

  /* Replacing double references with */
  /* single ones.                     */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used
        && t[m].br && t[m].ri != 0)
      t[m].li = t[m].ri-t[m].li;

  /* Building the clusters. */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      if (t[m].br && t[m].ri != 0) {
        /* A non-empty pair of     */
        /* parentheses is added to */
        /* a cluster.              */
        k = t[m].ri;
        t[m].ri = t[k].li;
        t[m].rc = t[k].lc;
        t[k].li = m;
        t[k].lc = 1; }

  /* Readjusting pointers. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used) {
      /* Updating all references */
      /* to the term. */
      while (t[m].lc) {
        /* Next element of the cluster. */
        k = t[m].li;
        t[m].li = t[k].ri;
        t[m].lc = t[k].rc;
        /* Now the pointer in the */
        /* term k is readjusted.  */
        t[k].ri = new_place;
        t[k].rc = 0; }
      new_place++; }

  /* Relocating all used terms. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used) {
      /* Relocating the term. */
      t[m].used = 0;
      t[new_place].li   = t[m].li;
      t[new_place].lc   = t[m].lc;
      t[new_place].used = 1;
      t[new_place].ri   = t[m].ri;
      t[new_place].rc   = t[m].rc;
      t[new_place].br   = t[m].br;
      new_place++; }

  /* Setting the marks off */
  /* and replacing single references */
  /* with double ones.              */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used) {
      t[m].used = 0;
      if (t[m].br && t[m].ri != 0)
        t[m].li = t[m].ri - t[m].li; }

  return; }
```

## 3.4 Elimination of Superfluous Passes

The above garbage collection algorithm performs 6 passes of the cell space. We can, however, reduce the number of passes, thereby increasing the efficiency of the algorithm at the expense of making it more difficult to understand.

First of all, double references can be replaced with single ones in the course of marking.

The inverse transformation can, similarly, be performed during the relocation of cells.

Thus we obtain a four-pass garbage collection algorithm. But the pass dealing with the readjustment of pointers can also be eliminated, although this can be achieved only by applying rather subtle tricks! The point is that this pass can be combined—as a whole—neither with the construction of clusters nor with the relocation of cells. And yet, its work can still be distributed over two other passes, so that it will be done partially during the construction of clusters and partially during the relocation of cells.

The key idea is due to Jonkers [Jon79, Coh81, CN83]. Let $C$ be a typical cell. Then a master of $C$ whose address is greater than that of $C$ will be said to be its “right” master, otherwise it will be said to be its “left” master.

During the construction of clusters, all left masters of $C$ (containing forward pointers to $C$) are added to its cluster. When $C$ is reached, all its left masters are already added to the cluster, therefore all forward pointers to $C$ can safely be updated (and all left masters of $C$ removed from the cluster).

As this pass continues, all right masters of $C$ (containing backward pointers to $C$) are added to its cluster.

Thus, at the end of this pass, the cluster of $C$ contains all its right masters, whereas all forward references to $C$ (contained in its left masters) have already been readjusted.

The next pass relocates the cells until $C$ is reached. Then all backward pointers to $C$ are updated (and all right masters of $C$ removed from the cluster).

The aforementioned “segregation” of the left masters of a cell from the right ones does enable us to circumvent the danger of corrupting the clusters. This is due to the fact that, when a left master of $C$ is relocated, the forward pointer to $C$ held in the master has already been updated (during the previous pass).

On the other hand, when a right master of $C$ is relocated, the backward pointer to $C$ held in the master has already been updated, as $C$ is reached before all its right masters.

## 3.5 A Three-Pass Algorithm

Now we can give a description of the three-pass garbage collection algorithm written in C.

```
void mark_term(address m) {
  address k, k_max;
  if (!t[m].used) {
    t[m].used = 1;
    if (t[m].br && t[m].ri != 0) {
      k     = t[m].li;
      k_max = t[m].ri;
      /* Replacing a double reference */
      /* with a single one. */
      t[m].li = k_max-k;
      for ( ; k<=k_max; k++)
        mark_term(k); }}
  return; }

void garb_coll(void) {
  address  m, k, new_place;

  /* The marking phase. */
  mark_term(MEMORY_LOW);

  /* Building the clusters. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used) {
      /* Readjusting all forward */
      /* references to the term. */
      while (t[m].lc) {
        /* Next element of the cluster. */
        k = t[m].li;
        t[m].li = t[k].ri;
        t[m].lc = t[k].rc;
        /* Now the pointer in the */
        /* term k is readjusted.  */
        t[k].ri = new_place;
        t[k].rc = 0; }
      if (t[m].br && t[m].ri != 0) {
        /* A non-empty pair of     */
        /* parentheses is added to */
        /* a cluster.              */
        k = t[m].ri;
        t[m].ri = t[k].li;
        t[m].rc = t[k].lc;
        t[k].li = m;
        t[k].lc = 1; }
      new_place++; }

  /* Updating backward pointers and */
  /* relocating the used terms.     */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used) {
      /* Updating all backward pointers */
      /* to the term. */
      while (t[m].lc) {
        /* Next element of the cluster. */
        k = t[m].li;
        t[m].li = t[k].ri;
        t[m].lc = t[k].rc;
        /* Now the pointer in the */
        /* term k is readjusted.  */
        t[k].ri = new_place;
        t[k].rc = 0; }
      /* Setting the mark off and */
      /* relocating the term.     */
      t[m].used = 0;
      t[new_place].li   = t[m].li;
      t[new_place].lc   = t[m].lc;
      t[new_place].used = t[m].used;
      t[new_place].ri   = t[m].ri;
      t[new_place].rc   = t[m].rc;
      t[new_place].br   = t[m].br;
      /* Replacing a single references */
      /* with a double one in */
      /* a non-empty pair of */
      /* parentheses. */
      if (t[new_place].br
          && t[new_place].ri != 0)
        t[new_place].li =
```

| Representation | Concatenation cost | Copying cost |
|---|---|---|
| Classical | low | high |
| Conciliatory | low | medium |
| Vector | high | low |

Figure 15: Peculiarities of different expression representations.

```
          t[new_place].ri
            - t[new_place].li;
      new_place++; }

  return; }
```

## 3.6 The Differences from Jonkers' Algorithm

It is evident that, as far as the compaction phase is concerned, the above three-pass algorithm bears a strong resemblance to Jonkers' compaction algorithm [Jon79]. Nevertheless, the two algorithms are not completely identical, since the same basic idea is used in different environments.

First of all, unlike Lisp S-expressions, the top level of a Refal expression is represented by an—arbitrarily long—sequence of adjacent cells, rather than by a single cell. Therefore, a Refal expression has to be referenced with two pointers, rather than with a single one. Moreover, the representations of two Refal expressions may partially overlap! This is not true of S-expressions, since only the whole top level of an S-expression can be shared.

The applicability of Jonkers' idea to Refal expressions is based on the fact that the cells that had been adjacent before the relocation remain adjacent after having being relocated, with their relative positions unchanged. This observation enables us to replace double references to subexpressions with single ones, and to forget, for some time, that each pointer, in fact, makes reference to a vector of cells, rather than to a single cell.

# 4 The Peculiarities of the Vector Representation

## 4.1 Efficiency of Different Representations

Which of the representations of ground expressions is the most "efficient"? It is evident that our desire to have fast copying inevitably comes into conflict with the desire to have fast concatenation (see Figure 15).

Thus different representations imply different styles of programming. Therefore, the choice of the "best" representation cannot be made on the basis of benchmarks obtained by running a few programs on several Refal systems based on different representations, because, if a program exhibits poor performance with respect to some Refal implementation, it may prove that the program is "ill-written", rather than that the implementation is "inefficient".

It is fairly easy to construct artificial example programs exhibiting behavior either favorable or unfavorable for the "vector" representation. For instance, consider the function TT:

```
<TT () tX>      =    tX
<TT (* eA) tX> =    <TT (eA) (tX tX)>
```

It can be easily seen that, in the case of the "classical" representation, the execution time of TT exponentially depends on the first argument's length. On the other hand, in the case of the "vector" or "conciliatory" representation, the execution time is proportional to the first argument's length.

## 4.2 How to Reduce the Cost of Concatenation

Since, in the case of the "vector" representation, the cost of concatenation is comparatively high, the question arises: does the cheap copying makes up for the loss in the speed of concatenation?

Fortunately, in many cases, actual concatenation is not needed and can be avoided. This will be shown in the following sections.

### 4.2.1 Special Cases of Concatenation

If expressions are represented by vectors, the concatenation of two expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ has to be performed by copying the top-level terms of the expressions. If $\mathcal{E}_1$ has the length $L_1$, and $\mathcal{E}_2$ the length $L_2$, a contiguous vector of the size $L_1 + L_2$ is requested, and the top-level terms of the two expressions are copied into that vector to produce a representation of the expression $\mathcal{E}_1\mathcal{E}_2$. Note that the subexpressions appearing in $\mathcal{E}_1$ and $\mathcal{E}_2$ inside parentheses do not have to be copied, as it is sufficient to copy only references to them.

In many cases, however, the cost of concatenation can be reduced.

1. If $\mathcal{E}_1$ is empty, the concatenation amounts to returning two pointers: to the beginning and to the end of $\mathcal{E}_2$.

2. If $\mathcal{E}_2$ is empty, the concatenation amounts to returning two pointers: to the beginning and to the end of $\mathcal{E}_1$.

3. If the vector representing $\mathcal{E}_1$ is situated in the memory just before the vector representing $\mathcal{E}_2$, the concatenation amounts to returning two pointers: to the beginning of $\mathcal{E}_1$ and to the end of $\mathcal{E}_2$.

4. If the vector representing $\mathcal{E}_1$ is situated in the memory at the very end of the heap, only the vector representing $\mathcal{E}_2$ has to be copied (into the place just after the representation of $\mathcal{E}_1$).

At first glance, case 3 seems to be unlikely to be met with in practice. Real programs, however, often traverse their argument expression by moving its top-level terms—one by one—from one variable to another. Thus both variables reference two adjacent parts of the same expression, and the concatenation amounts to readjusting pointers.

### 4.2.2 Functions with Explicit Arity and Co-Arity

In real programs most user-defined functions take a fixed number of inputs and return a fixed number of results. The number of inputs is usually called the *arity* of the function, and the number of outputs its *co-arity*.

Most current Refal implementations, however, require the arity and co-arity of all functions to be equal to 1. In other words, each function is supposed always to take one input and to return one result.

This limitation, traditionally, is circumvented by Refal programmers in the following way.

If a function `FUNC` has to be given $m$ input expressions $\mathcal{E}_1$, $\mathcal{E}_2$, ..., $\mathcal{E}_m$, these expressions are enclosed in parentheses and concatenated to produce a single input expression in the function call

$$\texttt{<FUNC (}\mathcal{E}_1\texttt{)(}\mathcal{E}_2\texttt{)}\ \ldots\ \texttt{(}\mathcal{E}_m\texttt{)>}$$

Then, if `FUNC` has to produce $n$ results $\mathcal{R}_1$, $\mathcal{R}_2$, ..., $\mathcal{R}_n$, the results are enclosed in parentheses and concatenated to produce a single result expression.

$$\texttt{(}\mathcal{R}_1\texttt{)(}\mathcal{R}_2\texttt{)}\ \ldots\ \texttt{(}\mathcal{R}_n\texttt{)}$$

It is evident that in both cases the real concatenation could have been avoided, if Refal enabled the programmer to specify the arities and co-arities of functions.

On the other hand, the arities and co-arities could be inferred by a Refal compiler automatically [Rom88, Rom90], in which case only the internal intermediate language of the Refal system would need to explicitly specify the arities and co-arities of functions.

### 4.2.3 Structuring Data with Parentheses

Concatenation becomes expensive only when applied to long expressions (recall that the *length* of an expression is equal to the number of the *top-level* terms of the expression). Thus keeping the length of expressions within reasonable limits—by rational use of parentheses—may reduce the cost of concatenation.

For example, suppose that a Refal program has to operate on stacks whose elements are ground expressions. Let the empty stack be represented by the empty expression, and the result of pushing an expression $\mathcal{E}_{top}$ on the top of a stack $\mathcal{E}_{stack}$ by the expression

$$\mathcal{E}_{stack}\ \ (\mathcal{E}_{top})$$

Then the stack containing expressions $\mathcal{E}_1$, $\mathcal{E}_2$, ..., $\mathcal{E}_k$ takes the form

$$(\mathcal{E}_1)(\mathcal{E}_2)\ \dots\ (\mathcal{E}_k)$$

where the stack's top is on the right. It is obvious that, as the stack size grows, pushing a new element on the stack becomes rather expensive, necessitating as it does the concatenation of the whole stack with the new element. However, if the result of pushing $\mathcal{E}_{top}$ on the stack $\mathcal{E}_{stack}$ were represented by the expression

$$(\mathcal{E}_{stack})\ \ (\mathcal{E}_{top})$$

then the problems caused by concatenation would be avoided, as pushing a new element on the stack would result in copying no more than two (top-level) terms.

## 4.3 Vector Representation and Virtual Memory

A virtual memory system is worth using only when memory accesses exhibit the property of *locality*, which means that memory accesses are concentrated—over a long period of time—in a few relatively small areas of the address space.

The property of locality is not usually satisfied in the case of list processing systems, as, in the course of time, the elements of lists become intermixed. Hence, logically adjacent elements cease to be adjacent in the address space. Besides, in systems with garbage collection, the lists tend to spread throughout the whole address space, with the really used cells being separated by vast gaps of free space. Thus, for example, if the main memory contains 1/10 of the lists, then 9 of 10 accesses to the lists result in the page fault [BM67].

Numerous garbage collection algorithms have been developed that try to counteract the harmful effects of "spreading" and "shuffling" by *linearizing* and *compacting* lists [Coh81, AT83].

The "vector" representation of expressions is likely to exhibit better performance in the virtual memory environment than the "classical" and "conciliatory" ones, as it represents logically adjacent terms by cells adjacent in the address space. Besides, the compacting garbage collection tends to reduce the size of the active address space.

## 4.4 The Copying Problem and Implementations of Refal

The high cost of copying prevented Full Refal [Tur71], Refal-4 [Rom87b, Rom87c]—and other similar projects—from being implemented. The implementation of Refal-5 [Tur89], based as it is on the "classical" representation, does not enable the programmer to use all resources of the language.

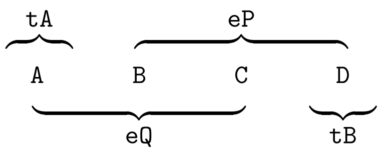


Figure 16: Overlapping variable values.

The implementation of Refal Plus, an extension to Refal-5, is based on the "vector" representation, which enables the user to write programs in functional style [GR91], and to make extensive use of the advanced features of the language. Consider, for example, the following function definition written in Refal Plus:

```
LR { eX, eX: tA eP, eX : eQ tB
                    = (eP)(eQ); };
```

We can see that the argument of a Refal Plus function can be matched against several different patterns—which is not true of programs written in Refal-2. Now, if we evaluate the call

```
<LR A B C D>
```

the variable `eX` takes the value `A B C D`. Then this value is matched against the two patterns `tA eP` and `eQ tB`, so that the expression `A B C D` is decomposed in two different ways: first, into the subexpressions `A` and `B C D`, and then into `A B C` and `D`.

Thus, in the case of Refal Plus (and Refal-5), the values of different variables may overlap (see Figure 16). Consequently, if a new expression has to be built from the—possibly overlapping—values of variables, the fragments of the variable values cannot be freely concatenated to form the result: they may have to be copied.

This problem, however, has never emerged in the implementations of Basic Refal [Tur71, Tur77] and Refal-2 [KR87, Rom87a], as those Refal dialects do not provide means for referencing overlapping fragments of the same expression via several variables, which has enabled the "classical" representation to be successfully used.

The "Refal Assembly Language" used as an intermediate language by the implementations of Basic Refal and Refal-2 [KRT72, Tur77, Rom87a], was tailored to the peculiarities of the "classical" representation, for which reason it is of little use in the implementations of Refal Plus. On the other hand, the language RL [Rom88] could be used as an intermediate language in the implementations of Refal Plus as well as in those of other Refal-like languages, such as FLAC [Kis87].

## 5 Conclusion

The “vector” representation of ground expressions allows the copying of expressions to be reduced to the copying of pointers, thereby enabling the Refal programs to be written in more functional style. Besides, this representation can serve as a basis for implementing advanced dialects of Refal, such as Refal Plus.

In addition, the “vector” representation consumes less storage space, as compared to those based on doubly-linked lists, eliminating as it does the references both to the previous cell and to the next one. In this respect the change-over to the “vector” representation is a more “revolutionary” step than the elimination of the reference to the previous cell only, as suggested by Mansurov and Ejsymont [ME87].

In the virtual memory environment, the “vector” representation should exhibit better performance than the representations based on doubly-linked lists, as the logically adjacent terms become adjacent in the address space, whereas the compacting garbage collector tends to reduce the size of the active address space.

## References

[AT83] Sergei M. Abramov and Svetlana B. Trubicyna. A linearizing garbage collection algorithm that minimizes the swapping of memory pages. In *Proceedings of the All-Union Seminar “Program Optimization and Transformation”*, volume 2, pages 5–12, Novosibirsk, 1983. The USSR Academy of Sciences, Siberian Division, Computing Center. (In Russian).

[BM67] D. Bobrow and D. Murphy. Structure of a Lisp system using two-level storage. *Communications of the ACM*, 10(3):155–159, March 1967.

[Chr66] C. Christensen. On the implementation of AMBIT, a language for symbol manipulation. *Communications of the ACM*, 9(8):570–573, August 1966.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[FOT69] Stanislav N. Florencev, Victor Ju. Oljunin, and Valentine F. Turchin. An efficient interpreter for the language Refal. Preprint 29, Keldysh

Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1969. (In Russian).

[GR91] Ruten F. Gurin and Sergei A. Romanenko. *The programming language Refal Plus.* INTERTEKH, Moscow, 1991. (In Russian).

[Hen80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[Jon79] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):26–30, July 1979.

[Kis87] Victor L. Kistlerov. The basic principles of the algebraic manipulation language FLAC. Preprint, Institute of Management Problems, Moscow, 1987. (In Russian).

[Kor86] Alexandr V. Korljukov. *Lecture Notes for the Students Taking the Special Course "Applied Algebra", Specialization 2013.* Grodno State University, Grodno, USSR, 1986. (In Russian).

[KR87] Andrei V. Klimov and Sergei A. Romanenko. *Programming System Refal-2 for ES Computers: The Source Language.* Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1987. (In Russian).

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[KRT72] Andrei V. Klimov, Sergei A. Romanenko, and Valentine F. Turchin. *A Compiler for the Language Refal.* Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1972. (In Russian).

[ME87] Nikolai. N. Mansurov and Leonid. K. Ejsymont. An implementation of the extended language Refal based on single-linked lists with ring chains. Preprint 20, Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1987. (In Russian).

[Mor78] F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, August 1978.

[Rom87a] Sergei A. Romanenko. *Implementation Techniques for the Language Refal-2.* Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1987. (In Russian).

[Rom87b] Sergei A. Romanenko. Refal-4, an extension to Refal-2 for representing results of driving. Preprint 147, Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1987. (In Russian).

[Rom87c] Sergei A. Romanenko. Driving for programs written in Refal-4. Preprint 211, Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1987. (In Russian).

[Rom88] Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprizingly natural and understandable structure. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[Rom90] Sergei A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[Tur71] Valentine F. Turchin. Programming in the language Refal. Preprints 41, 43, 44, 48, 49, Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences, Moscow, 1971. (In Russian).

[Tur77] Valentine F. Turchin, editor. *Basic Refal and its Implementation on Computers.* GOSSTROJ SSSR, CNIPIASS, Moscow, 1977. (In Russian).

[Tur79] Valentine F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

[Tur86] Valentine F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Tur89] Valentine F. Turchin. *Refal-5. Programming Guide and Reference Manual.* New England Publishing Co., Holyoke, Massachusetts, 1989.

[Wei63] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–536, September 1963.