Interpreter Design Considerations

Dick Grune
dick@dickgrune.com

November 3, 2014; DRAFT

Contents

1 Introduction

The postscan yields the Teckel program as a binary tree with a slightly more convenient syntax than is produced by the parser.° The system treats the tree as a functional program, and continuously applies reductions to its nodes, until it has been reduced to the empty program. The node undergoing reduction is called the "redex". During these reductions the `\Toutput` command(s) will have produced the desired result(s).

° construct this syntax

2 Redex Identification

Tree reduction raises the question of redex identification. If all sets were finite there would be no problem: the search tree would be finite and any redex identification order would be fine. But many sets are infinite; $\mathbb{N}$ is the first example. Now given the purpose of Teckel the set of integers could be limited to say 1..100, but infinite sets also originate from recursive definitions:

$$S \equiv xS;\ S \equiv \varepsilon$$

So we need to be able to handle infinite sets.

This leaves us with an infinite search tree, which might well contain infinite subtrees that do not contain a solution, and we need means to prevent Teckel running around like mad trying to cover the full extent of such an infinite tree. Breadth-first search seems the only option. Note that even AND nodes need to be searched breadth-first: one of the branches may loop forever while another one comes up with the answer *false*. Breadth-first search will then find the answer in finite time.

2.1 Breadth-first Search of the Program Tree

To simplify matters we require the search tree to be a unary/binary tree: each node has 0, 1, or 2 children. This means for example that a compound filter like $\{a, b \in \mathbb{N} \mid a^2 = b\}$ must be decomposed, as for example shown in Section 3.3.

Each turn of the graph reduction machine starts anew finding a redex, as follows. The redex finder ("the teckel") starts at the top of the tree. If the node has one child, it goes to that child. If the node has two children, and if it went to the left child last time the node was visited by the redex finder, it now goes to the right child, and vice versa. Eventually it finds a node without children; this is the redex. This procedure seems to distribute the available processing power fairly over the tree, and guarantees that every node will eventually be visited.

The same could probably be achieved by keeping all nodes in a linear list, declare the head of the list to be the redex, and add new nodes at the end. But quite often entire subexpressions have to be discarded, and finding their nodes in the list would be very problematic; in the present model the tree in question can just be released. Also the Teckel model allows one or both of the child pointers to be used as data path, rather than as teckel path; the last model would make that much harder.

The presented model is simple, elegant, and does all we want, with minimal means, but it is quite difficult to program. The main problem is the absence of context information: when reducing a node, all pertinent information must be available from inside that node, which means that an earlier visit must have stored it there. So the context is there but it is hidden in conventions between nodes. It is of paramount importance to keep these conventions as clean and as explicit as possible.

We do not want to use parallel tasks, to avoid reproducibility problems.

2.2 A Characterization of the Execution Order

If the program happens to be a balanced tree with exactly 2^n leaves, the execution order can be characterized as the binary reverse of the step number:

step	bin.	rev.	node#
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

For what it is worth. See `revbin.c` for a demo.

2.3 Global Data of the Teckel

The address of the node identified as the redex is stored in the global variable `Node`, and the address of the node where the teckel was before it found the redex is in `Parent`; the top of the program is kept in the constant `Root`.

Since more than one part of the program can be interested in a node, several pointers can point to `Node`, the teckel can reach it from several sides, and `Node` can have more than one parent. The present node `Parent` is just the node that the teckel happened to pass last. So we cannot update "the parent" of a node. We can, however, draw the teckel's attention to Parent (by setting its `is_teckel_stop` bit). The next time the teckel comes around it will stop at the marked node and perform the update. When the teckel approaches Node through another path, it will again set the `is_teckel_stop` bit of the then Parent, and later again the teckel will also update that node.

2.4 Local Data of the Teckel

Three fields in the node control the route of the teckel. One, `is_teckel_stop`, tells the teckel whether or not to stop at this node. If the node is not a teckel stop, the teckel finds before it two possible exits, guarded by a flap door which allows access to one of the exits only; the field `flap_dir` indicates which. A third field, `flap_is_fixed`, tells the teckel whether or not to change the position of the flap door. The assignment

```
n->flap_dir ^= (n->flap_is_fixed^1)
```

then updates the field `flap_dir`.

Three types of nodes can be distinguished in the program tree/dag: those which are to be passed by by the teckel; those that require the attention of the teckel; and those that are finished and done. And on each path from the Root we find them in that order.

Each node has a local field `status`, with the values `NEW`, `WORK`, and `DONE`. Together with the values for `is_tecke_stop` there are six combinations:

	`is_teckel_stop`	
`status`	1	0
`NEW`	initialize	✘
`WORK`	work	pass
`DONE`	alert Parent	✘

where ✘ indicates a situation that cannot occur: NEW nodes start with `is_teckel_stop` set to 1, and in a DONE node there is no way for the teckel to continue. When the initialization or the work is done, the teckel must set its administration fields appropriately for its next visit. Normally this is done by calling the routine `bye()`, which willdetermine the number and position of teckel-needing (`status != DONE`) children, but sometimes there are other considerations arising from limitations in the basic functional model.

The basic functional model describes a world of expressions which are evaluated bottom-up, and in which we are only interested in the single complete finished result. In practice there are several situations in which we want to see or examine a partial result, i.e. the result of one of the children, while the other is still being computed.

- There are several node types in which it is useless to compute one subtree before the result of the other is known. Examples are: the if-then-else node, in which no progress can be made until the condition has been computed; and the limited output statement, which cannot do useful work until the limit is known.
- It is often profitable to examine a finished operand, while the other is still being computed. Prime example is the Boolean $\wedge$: when one operand turns out to be *false*, the entire computation of the $\wedge$ can be abandoned, but one can also think of multiplication by 0, etc.
- Sets can be infinite: we cannot wait for them to be computed completely, and in a `\Toutput` statement they have to be displayed value by value. This requires yielding on the one hand one finished value, and on the other hand keeping the non-finished rest of the set.

It turns out that if one operand has precedence over another, it is always the left operand that needs to be computed first. This situation is handled by a special version of the `bye()` routine, `bye_ignore_right_child()`.

The case that a node is interested to see a partial result is so general, that any node that sets its `status` to `DONE` must call `alert_Parent()` at the end, which will just set `is_teckel_stop` to 1, rather than call `bye()`.

So the routine `reduce_T()` for reducing `Node` with node type `R_T` or `T` will in general look something like shown in Figure 1. We do not need to check for `Node->status = DONE`; code in the basic loop in `reduce.c` catches that case, alerts the parent and restarts the teckel.

```
static void
reduce_T(void) {
    /* T: syntax rule */

    /* optional sanity check(s) for a node of type T */
    if (Node->child[0] == 0) oops("no left child to T", Node);

    if (Node->status == NEW) {
        /* initialization code for a node of type T */
        Node->status = WORK;
        bye();
        return;
    }

    /* Node->status == WORK; reasons for stopping here are: */
    /* 1. left child is finished */
    if (Node->child[0]->status == DONE) {
        /* code to process Node->child[0] */
        alert_Parent();
        return;
    }
    /* 2. right child is finished */
    if (Node->child[1]->status == DONE) {
        /* code to process Node->child[1] */
        bye();
        return;
    }

    oops("teckel stopped for no valid reason", Node);
}
```

Figure 1: Skeleton routine for reducing `Node` of type `T`

3 Sets

Sets are complicated and difficult to implement, for the following reasons:

- Sets are unlike the other data types in Teckel in that they can be of unbounded size. For such objects to be useful, it is necessary that their element values become available as soon as possible, and certainly before the whole set is done.
- Set elements are restricted by two mechanisms:
  - their values must be unique;
  - the values used in producing them must come from specified sets and have passed a (possibly dummy) filter.

  Data to perform the required tests must be available at all times.

- Sets can be implemented as either lazy or strict. The choice has great impact on the ease of generation and the ease of use by the implementer (the user should not be affected by the choice).

The three issues, expression termination management, value uniqueness, and lazy versus strict, are strongly interrelated, so careful analysis is needed.

3.1 Types of sets

There are basically four types of sets:

- Enumerated sets: $\{\}$, $\{42\}$, $\{1, 9, 4, a^2\}$; the elements are explicitly specified by expressions.
- Expression sets: $\{abc, a \in V\}$, $\{(a^2, b), a \in \mathbb{N}, b \in Q | a \mod 2 = 0\}$; an expression for the elements is given, the variables in which are supplied by sets.
- Pattern sets: $\{(a, 2) \in \Theta\}$; a pattern expression for the elements is given, which must match elements from a given set.
- Range sets: $\{0 < i \leq 10\}$.

All sets except enumerated set may be filtered through a condition, f.e. $\{0 < i \leq 10 \mid i \mod 2 = 0\}$.

Three of the four set types are based on a generator. Enumerated sets are different and are dealt with in Section 4.2.

3.2 Erratic Expression Termination

A set element expression can
- 1a. terminate with a value;
- 1b. terminate late with a value;
- 2a. terminate with `NO_RESULT`;
- 2b. terminate late with `NO_RESULT`;
- 3. fail to terminate.

An expression is considered to "terminate late" if it terminates later than an expression that comes textually after it. This can easily happen, even though these later expressions get exponentially fewer teckel runs.

The results of late expressions must be made available to processes that may be waiting for the result of the first expression. Failure to terminate should not stop the other set element expressions from getting evaluated.

Expressions that terminate with `NO_RESULT` must be eliminated from the set. If all members of a finite set yield `NO_RESULT`, the set becomes empty.

Infinite sets don't terminate. This can happen even if their size is finite: the set $\{n - n, n \in \mathbb{N}\}$ is an example of a set of finite size that takes forever to compute. Finite sets fail to terminate if at least one of its members fails to terminate.

3.3 Filtering

The most general form of a filtered expression set is

$$\mathcal{S} \equiv \{\mathcal{F}(a_1, ..., a_n), a_1 \in A_1, ..., a_n \in A_n \mid \mathcal{T}(a_1, ..., a_n)\} \quad (1)$$

where $\mathcal{F}$ is the function that yields the elements and $\mathcal{T}$ is the test in the filter. This is far too complicated to implement, so it has to be decomposed.

There are (at least) two ways to decompose (1). The first reduces it to a sequence of forms $\{\mathcal{F}(x), x \in \mathcal{Z}\}$, each of which correspond to the strict set implementation in Section 4, as follows:

$$\begin{aligned}
&\mathcal{S} \equiv \{F_{n-1}(a_1), a_1 \in A_1\}\\
&F_{n-1}(a_1) \equiv \{F_{n-2}(a_1, a_2), a_2 \in A_2\}\\
&\ldots\\
&F_1(a_1, ..., a_{n-1}) \equiv \{F_0(a_1, ..., a_n), a_n \in A_n\}\\
&F_0(a_1, ..., a_n) \equiv \mathcal{F}(a_1, ..., a_n) \textbf{ if } \mathcal{T}(a_1, ..., a_n)
\end{aligned}$$

and the computation of $F_0(a_1, ..., a_n)$ evaluates $\mathcal{F}(a_1, ..., a_n)$ only if $\mathcal{T}(a_1, ..., a_n)$ yields *true*.

Actually this is not correct: if $\mathcal{F}$ is of type `T`, $\mathcal{S}$ ends up being of type (set of)^n `T`, rather than of type set of `T`. To correct his, we need a special mechanism to combine the sets produced by expressions $F_i(a_1, ..., a_{n-i})$ for $0 \leq i \leq n-1$ into a set of the same type.

The second way to decompose (1) is by using the $\bigcup$ operator:

$$\begin{aligned}
&\mathcal{S} \equiv \bigcup_{a_1 \in A_1} F_{n-1}(a_1)\\
&F_{n-1}(a_1) \equiv \bigcup_{a_2 \in A_2} F_{n-2}(a_1, a_2)\\
&\ldots\\
&F_1(a_1, ..., a_{n-1}) \equiv \bigcup_{a_n \in A_n} F_0(a_1, ..., a_n)\\
&F_0(a_1, ..., a_n) \equiv \{\mathcal{F}(a_1, ..., a_n) \textbf{ if } \mathcal{T}(a_1, ..., a_n)\}
\end{aligned}$$

This is type-correct. It may be noted that the intuitive implementation of the $\bigcup$ operator is quite similar to the strict set implementation in Section 4.˚ ˚ is that so?

Both decompositions suggest not checking for uniqueness until the final phase. This reopens the discussion about the who and when of uniqueness testing.˚ ˚ uniqueness

Both the above constructions seem amenable to eager testing (Section 12). testing again

3.4 Lazy versus Strict Set Implementation

A lazy implementation hands out the unevaluated expression of a set element to the requesting operation. A strict implementation makes the the requesting operation wait until an evaluated element (not necessarily the sequentially first!) becomes available.

Theoretically lazy evaluation is preferable and also fits in well with the rest of Teckel. But when expressions are handed out unevaluated, the system cannot prevent multiple identical values from emerging: $\{2+2, 1+3\}$. So operations sensitive to multiple identical values will have to do a uniqueness check; others may accept multiple identical values, but this will probably waste CPU cycles.

A strict implementation will hand out only fully evaluated unique values, but has to be careful not to get caught in non-terminating computations.

One may wonder whether this a a problem at all: the only requirement is that the user cannot see the difference. Also, lazy evaluation is usually deployed to combat infinite loops, but with our breadth-first evaluation of the program tree these have ceased to be a problem.

Now the only thing the user sees is the output, so it seems to boil down to the requirement that *output* sets values are unique. But that is not entirely true. The operator `\in` does not mind if it gets a value twice: it just stops at the first occurrence; binary operators like $\wedge$ just waste time on them; but the size operator $|V|$ gives a wrong answer.

So in a strict implementation any operation yielding a set has to do a uniqueness check; in a lazy implementation some specific routines have to do one. For the rest it does not matter much: a strict implementation may waste cycles by computing an element that will not be needed but it will not prevent all solutions from being found. It may, however, prevent a program from terminating.° And a lazy implementation may waste cycles on computations on repeated elements.

° check

So far the producer's side, but there is also the consumer's side. And lazy and strict sets differ greatly in their ease of use.

We will first describe the set implementation for lazy and strict sets, and then examine the operations involved in sets, and see how they are influenced by lazy or strict set implementation.

3.4.1 Lazy Set Implementation

Lazy sets are surprisingly simple to implement. When visiting a spine node for the first time, just set `status` to `DONE`: next time the teckel will stop and yield its left child, the unevaluated set element expression, as the value. The result is a set with possible duplicates.

Lazy sets are hard to use: each consumer has to trouble oneself with code to handle irregular expression termination, and value uniqueness if it is essential to the consumer.

3.4.2 Strict Set Implementation

Strict sets are much more complicated to implement, since all the problems of expression termination management and value uniqueness have to be solved in the nodes along the spine.

The question arises who is responsible for maintaining the uniqueness of the values, the set implementation or the supplier of the values? But the supplier cannot always know: if a set is filled with values computed as $F(a_i)$ for various unique values of a_i, these values can still contain duplicates, because there is of course no guarantee that $F(a)$ will yield different values for different values of a. An end-to-end argument says that since it is impossible to always avoid inserting a duplicate value in a set, it is the responsibility of the set itself to check, and nobody else needs to bother. ("Since not everybody can bother, only the last man needs to bother.") So the set, named or anonymous, must take care of value uniqueness, and retain a collection values computed so far.

The implementation is discussed in detail in Section 4.

Strict sets are very easy to use: all a consumer has to do is to send the teckel to the type 2 node of the set and wait until it comes back with a value.

3.5 Set Operations

There are the following set-yielding operations:

enumerated set	$\{1, 2, 1+1\}$
expression set	$\{a^2, a \in S \mid a \mod 2 = 0\}$
set union	$a \cup b$
set intersection	$a \cap b$
set difference	$a \setminus b$
distribution of operations over sets	$\{1, -1\} \times \{2, -2\}$
serial set union	$\bigcup_{a \in A} F(a)$

There are the following relational operations on sets:

set (in)equality	$=, \neq$
(proper) subset	$\subset, \subseteq$
(proper) superset	$\supset, \supseteq$

There are the following additional operations acting on sets:

membership	$3 \in \{2+1\}$, $3 \notin \{2+1\}$
size	$\lvert V \rvert$
output	`\Toutput{\{1, 1\}}`

And there are three set constants: $\mathbb{N}$ ($\{1..\infty\}$), $\mathbb{Z}$ ($\{0..\infty\}$), and $\emptyset$ ($\{\}$).

For a complete analysis of the problems we need to consider behavior and termination in the presence of

1. infinite sets;
2. expressions terminating in `NO_RESULT`;
2. non-terminating expressions

Negative results over infinite sets (set difference, negative membership) can only be obtained through a non-decreasing property (*NDC*) (Section 13). Exploitation of a non-decreasing property requires the values of a set to be produced in a controlled order. This is difficult but seems doable in a strict implementation. [A lazy implementation has no control over the order: unevaluated elements are handed out and go off with their users, losing any coherence. It may be possible to create enough dependency relations in the program tree to allow exploitation of non-decreasing properties, but it would be exceedingly difficult.]

3.5.1 Enumerated Sets

Syntactically this is the fundamental case; its implementation is a special case of the expression set. The main difference between the lazy and the strict implementation of an enumerated set is that the lazy implementation can hardly implement value uniqueness.

3.5.2 Expression Sets

The general form is $\{F(a), a \in A \mid T(a)\}$. Even if A does not contain duplicates, this operation needs to check for duplicates in the result: $\{a^2, a \in \{2, -2\}\} = \{4\}$.

The strict implementation is described in Section **??**.

For the generator $\{F(a), a \in \{a_1, a_2, ...\} \mid T(a)\}$ a lazy set implementation creates –almost by definition– the set

{
if $T(a_1)$ **then** $F(a_1)$ **else** `NO_RESULT` **fi**,
if $T(a_2)$ **then** $F(a_2)$ **else** `NO_RESULT` **fi**,
...
}

If $F(a_i)$ or $T(a_i)$ yields `NO_RESULT` the element i evaporates. If $F(a_i)$ or $T(a_i)$ fail to terminate the computation of element i fails to terminate. Again duplicate checking will have to be done elsewhere.

3.5.3 Set Union

Basically set unions just assembles its operands into one set, regardless of finiteness of termination. Lazy implementation can leave it at that, but the strict implementation will have to weed out the duplicates, essentially by using the general set implementation scheme (Section **??**).

There seems to be no simple set theoretical implementation of set union:

$$A \cup B \equiv \{??, a \in A, b \in B \mid ??\}$$

3.5.4 Set Intersection

Intersection of strict sets needs to keep copies of both its arguments as obtained so far. Apart from that there is no problem: it starts with two sets without duplicates and intersecting them cannot create duplicates.

Intersection of the lazy sets $\{a_1...a_m\}$ and $\{b_1...b_n\}$ produces $m \times n$ elements of the form **if** $a_i = b_j$ **then** a_i **else** `NO_RESULT` **fi**. If $a_i = b_j$ and a_i occurs p times and b_j occurs q times the element a_i will be entered $p \times q$ times. If the sets are infinite, the breadth-first redex identification (Section 2) makes sure that elements from both operands will be requested about equally.

Intersection can temporarily be implemented as

$$A \cap B \equiv \{a, a \in A \mid a \in B\}$$

at a loss of efficiency.

3.5.5 Set Difference

Set difference cannot take any decision until it has seen enough of its right operand. If that operand is finite, it has to see all of it; if it is infinite, some non-decreasing property has to be established (see Section 13). In both implementation styles these values must first be collected. Processing can then continue, either directly of as a set of equation as above.

Set difference can temporarily be implemented as

$$A \setminus B \equiv \{x, x \in A \mid x \notin B\}$$

at a loss of efficiency.

3.5.6 Distribution of Operations over Sets

Like intersection, strict implementation of distribution over sets needs to keep copies of both its arguments as obtained so far. But unlike intersection, duplicates can arise, as the example $\{1,-1\} \times \{2,-2\} = \{-2,2\}$ shows. So the general technique from Section **??** must be applied.

Lazy distribution over the set expression $\{a_1...a_m\}\xi\{b_1...b_n\}$ produces $m \times n$ elements of the form $a_i \xi b_j$, and there is no telling if duplicates arise. Also, if $a_i = b_j$ and a_i occurs p times and b_j occurs q times the element $a_i \xi b_j$ will be entered $p \times q$ times.˚ ˚ the point is?

It is interesting to see that intersection $\cap$ is a special case of distribution of the operator $\cap'$ over sets, with $a \cap' b \equiv$ **if** $a_i = b_j$ **then** a_i **else** `NO_RESULT` **fi**.

Operator distribution over sets can probably most easily be implemented as

$$A\xi B \equiv \{a\xi b, a \in A, b \in B\}$$

in the general case, although a more efficient implementation seems possible for the built-in operators.

3.5.7 Big set union

Set union ($a \cup b$) unifies sets: $\{1,2\} \cup \{3,2\}$ is OK, but $1 \cup \{3,2\}$ is not; it has to be rewritten as $\{1\} \cup \{3,2\}$. Likewise, although $\sum_{i=1}^{i=3} i^2$ means $1^2 + 2^2 + 3^2$, $\bigcup_{i=1}^{i=3} i^2$ does not mean $1^2 \cup 2^2 \cup 3^2$ but $\{1^2\} \cup \{2^2\} \cup \{3^2\}$. And in the same vein

$$\bigcup_{a \in \{a_1,a_2,...\}} F(a) \quad \text{means} \quad \{F(a_1)\} \cup \{F(a_2)\} \cup ...$$

The implementation of the $\bigcup_{a \in \{a_1,a_2,...\}} F(a)$ operation is almost identical to that of $a \in \{a_1, a_2, ...\}$, except that before allowing the public access to a, it is passed through the function F.

3.5.8 Big set intersection

The semantics of big set intersection is weird: $\bigcap_{a \in \{a_1,a_2,...\}} F(a)$ is the singleton $\{a\}$ if $a = F(a_i)$ for all a_i, and the empty set otherwise.

3.5.9 Set comparison

Both operands are evaluated in non-strict fashion. If some operands are finite or are non-decreasing answers may be obtained. If both sets are infinite without a non-decreasing property the comparison will not yield an answer. Such sets cannot be compared by algorithmic means.

Implementationwise the simplest set comparison is $a \subseteq b$: go through all members of a and check each element for presence in b. The following table shows what results can be obtained, depending on whether a and b are finite, non-decreasingly infinite (NDC), or just infinite.

$a \subseteq b$	b is finite	b is NDC	b is infinite
a is finite	pos & neg	pos & neg	pos
a is NDC	always neg	neg	–
a is infinite	always neg	neg	–

The entry "always neg" means that the negative answer is always obtained in finite time (except when a is finite but does not terminate).

The operation $a \subset b$ is almost the same but additionally requires b to contain at least one member not in a. So we have to go through b to find such a member. (If both a and b are finite a simpler solution is available: compare the lengths of a and b.)

$a \subset b$	b is finite	b is NDC	b is infinite
a is finite	pos & neg	pos & neg	pos
a is NDC	always neg	neg	–
a is infinite	always neg	neg	–

Set equality is usually tested in software in $O(n\ logn)$ time by sorting both sets and then comparing them by a linear sweep. This is not an option in Teckel. The simplest way seems to be to compute $a \subseteq b \wedge b \subseteq a$; for a and b finite the second condition can be replaced by $|a| = |b|$.

$a = b$	b is finite	b is NDC	b is infinite
a is finite	pos & neg	always neg	always neg
a is NDC	always neg	neg	neg
a is infinite	always neg	neg	–

Superset operations are the reverse of subset operations: $a \supset b \equiv b \subset a$.

3.5.10 Membership

The operation $e \in A$ is strict in its first –non-set– operand. The strict set implementation then checks each successive set value. For $e \in \{a_1..a_n\}$ the lazy implementation creates an expression $e = a_1 \vee ... \vee e = a_n$. If A is infinite and does not contain e, the operation does not terminate unless a non-decreasing property can be found.

The same holds for $e \notin A \equiv \neg(e \in A)$.

Intuitively negative membership seems harder than positive membership, but that is not true. If A is finite or the answer is yes, $e \in A$ will succeed in finite time; otherwise a non-decreasing property is required for termination. If A is finite or the answer is no, $e \notin A$ will succeed in finite time; otherwise a non-decreasing property is required for termination.

3.5.11 Size

The operation $|A|$ is strict in its operand. If A is infinite the operation does not terminate. The strict set implementation is OK; the lazy one will have to collect all values and remove duplicates, in a process similar to that of the strict implementation.

3.5.12 Output

Output of a set is a special case, because the set may be infinite, or just slow in being computed, and the user wants to see every available bit of it right away. To avoid confusion a set output statement prints the whole set available so far upon arrival of each newly computed value. This forces a set output command to retain access to the whole set computed so far.

This is no problem for the strict set implementation, but the lazy one will have to 1. weed out duplicates; 2. keep all previous results. The easiest way to do so, is to use code similar to the strict set implementation.

3.6 The set constants $\mathbb{N}$, $\mathbb{Z}$, and $\emptyset$

Although theoretically these are sets, not sequences, the elements of the first two are in practice guaranteed to be generated in sequential order. Both strict and lazy implementations can immediately satisfy access to the next element. The $\emptyset$ is trivial.

3.7 Access to a set value

The access of the value in a set element is as follows. A set element E with a value V is a value. When the teckel meets E, it makes the parent P a teckel stop. When the teckel then meets P, it obtains the value of the set element by copying (or linking) V, and then perhaps moves to the next element.

This means that node P has to know it interfaces between the set and a consumer. The only constructs that are clients of set values are the usual suspects known from Section 3.5:

(enumerated sets)
expression set
union
(intersection)
(difference)
(distribution of operations over sets)
membership
size
output

where the ones between parentheses either can be implemented using expression sets, or do not consume sets, although they produce them.

The remaining five constructs are all programmed specifically, and it can be assumed that they all use the same kind of interface node, say a "set element access" node. So a spine node of a set can be sure its Parent is a set element access node, and nothing else.˚ ˚ not true!!

How this affects further design is not yet clear.

3.8 The Set as a Value

Some expressions require the whole set as a value:

$$(1, \{1 + 1, 3 + 5\})$$

Other examples are the sets in a set of sets S; to ensure uniqueness of the values in S its elements must be compared for equality among each other.

Sets can be evaluated in strict fashion by requesting every element.

3.9 Conclusion

Although a lazy set implementation would probably be preferable on theoretical grounds, it suffers from the following drawbacks:

1. it cannot readily suppress duplicate values, which will continue to cause implementation problems and waste cycles;
2. the set size and output operators have to resort to strict set implementation anyway;
3. it prevents the implementation of non-decreasing properties; this problem is especially harmful, since if Teckel ever wants to obtain negative results over infinite sets, it will need to use non-decreasing properties;
4. even if the consumer process can afford to ignore duplicate values, it still has to deal with irregular termination, duplicating part of the strict implementation.

So set implementation should be strict. This means that the set implementation code keeps all values to itself; the system should lead the teckel to the Type 2 node of the set when an expression needs the next set value.

4 Strict Set Implementation

Unlike lazy implementation, the strict implementation does not allow the client access until the expression has been evaluated. So the strict set implementation has to worry about value uniqueness, about expressions not terminating, and about expressions terminating late, while at the same time making the first evaluated element available to the client(s).

4.1 The "Set" Data Structure

The following implementation fulfils these requirements.

The elements of a 'set of T' are not of type T but of type 'element of set of T' or equivalently 'T set element': this type allows two operations: `Value` and `NextElement`. In fact a *set element* is a node with left child `Value` and right child `NextElement`. The most natural set implementation is a chain of such nodes, the *spine*. Four segments can be distinguished in the spine:

1. The values that have already been computed; anybody may point to these; they are teckel stops.
2. An administration node; anybody in need of the next element points to here. It is not a teckel stop, unless it has been alerted by its right-hand child.
3. Elements whose values have not yet been computed; nobody is allowed to point to these (except their parents in the spine). They may have been specified directly in the program or may have been generated by a generator further on. They are not teckel stops.

4. The generator with its pertinent data: the generator function F, its parameter x, and the set the values of x are to be taken from. The generator is generally not a teckel stop, except initially or when it is an enumerated set or one of the set constants from Section 3.6.

This set-up is shown in Figure 2.

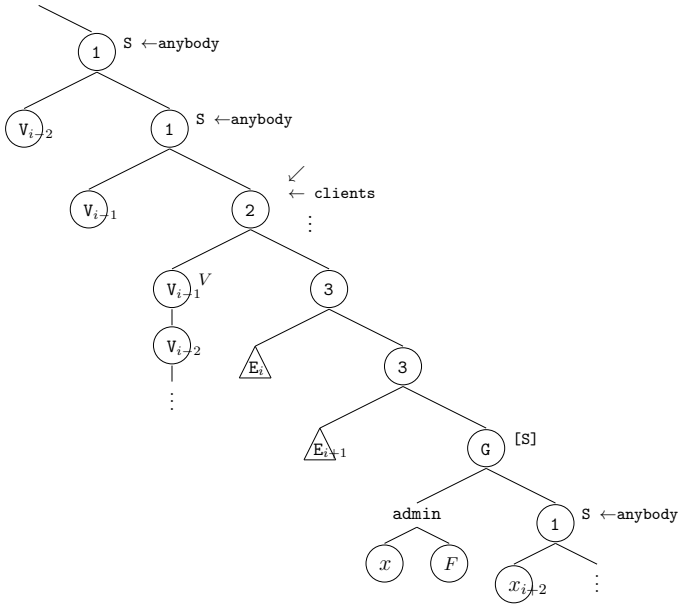


Figure 2: Data structure for sets

Here the numbered nodes represent Type 1, Type 2, and Type 3 nodes; `V`-s are finished values; `E`-s are expressions being computed; and `G` is the generator node. V is the list of values already computed, kept for uniqueness testing; unlike shown in the picture, the values V_{i-1}, V_{i-2}, etc., are not copies of but pointers to the corresponding values dangling from the Type 1 nodes. Note that these pointers do not cause loops in the program graph. The arrows next to the Type 2 node represent one or more clients waiting for the next value in the set; there may be other links to Type 1 nodes from outside, but there cannot be other links to Type 3 nodes or the generator, except when it is the only node.

The Teckel mechanism ensures that each of the Type 2 and Type 3 nodes and the generator get an exponentially decreasing amount of the processing power. (Section 2.)

Note that the set-up of the generator with another set as input implements a kind of semi-strict multiple-variable set: no set on a certain level starts until there is at least one value available from the set one level below. This does not impair generality but simplifies the implementation.

4.2 Enumerated sets

Although the scientific mind would like to unify the enumerated set with the other forms of sets by providing it with a dummy generator and a dummy filter,

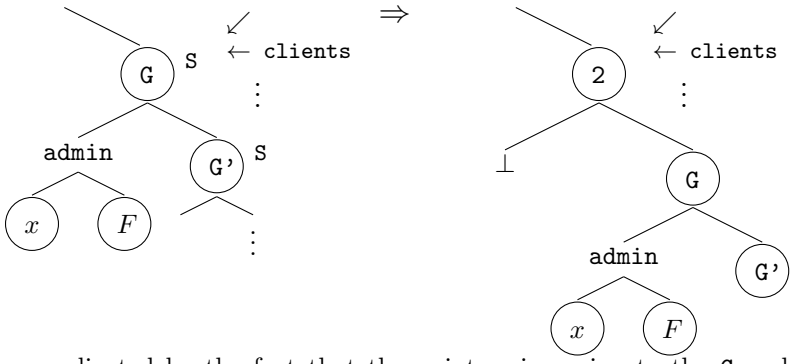
the syntactic form of the enumerated set is so similar to Figure 2 that common sense dictates to just leave it alone. Making the `R_enumerated_set` node a Type 2 `R_set_element_list` node, making the other node Type 3 nodes, and adding a terminator suffices to make the enumerated set conform to Figure 2.

So enumerated set do not have generators, but they can have Type 1, 2, and 3 nodes.

Also, the chain of genertors in a multi-variable set definition must stop somewhere. It can stop at one of the set constants from Section 3.6, or indeed at an enumerated set.

4.3 Run-time scenario

Initially the generator is the only representative of the set; it is a teckel stop, and its status is `NEW`. When the teckel reaches it for the first time it cannot alert its parent for anything, since its parent is some unknown client; so it has to do everything itself. It sets its status to `WORK` and inserts a Type 2 node in front of itself:



This is complicated by the fact that the pointers incoming to the `G` node must afterwards point to the Type 2 node. This requires fancy footwork.

The next time the teckel stops at the generator the status is no longer `NEW`. The teckel detaches the generator body and splits it into a one-element 'head' and the rest, the 'tail'. Next it makes a copy of itself, attaches the head to itself, attaches the tail to the copy, and attaches the copy as its right child. Then it turns itself into a Type 3 node and is no longer a teckel stop. Its left child is now an expression to be evaluated.

When the expression is completely evaluated, the Type 3 node is alerted and becomes a teckel stop. The expression can result in a value or in `NO_RESULT`.

When the teckel stops at a Type 3 node whose expression is finished, it alerts its parent, which can only be a Type 3 or Type 2 node, since nobody else can point to a Type 3 node.

When the teckel stops at a Type 3 node N whose expression is *not* finished, its right child must be a Type 3 node C whose expression *is* finished. N swaps its expression with that of C, removes C's stop bit, and makes itself a teckel stop. Now the situation is as if N own expression had finished. N can now alert its parent (an optimization) or wait for the teckel to come along the next time.° ° design decision

After a while the teckel will stop at a Type 2 node N whose right child C has an expression that is finished. This expression can be a value or `NO_RESULT`. If the expression is `NO_RESULT`, N can easily remove C by unlinking it from the

chain, and is no longer a teckel stop. The removing can always be done since nobody can be pointing to C, and there will always be another node under C.

If the expression is a value, it needs to be compared to the values in V. This is a finite task that can be performed by the teckel on the spot (even if the elements of the set are of type 'set'; see Section 3.8). If the test fails, C is removed as before, and N is no longer a teckel stop.

If the value is new, N adds it to its element list V, swaps V with the value of C, changes itself to a Type 1 node and its right child C to a Type 2 node. The Type 1 node is a teckel stop, the Type 2 node is not.

At a certain moment the generator will find it is out of elements; it then alerts its parent, which can be a Type 3 or a Type 2 node, and nothing else.

If the teckel stops at a Type 3 node because its right child C is an exhausted generator, it discards the left child of C (its generation body) and changes its type to "terminator".

If the teckel stops at a Type 2 node because its right child C is an exhausted generator, it discards C and V, and turns itself into a terminator (which is a teckel stop).

If the teckel stops at a spine terminator it alerts its parent, which can be a Type 3 node, a Type 2 node, or a client.

If a Type 3 node finds its right child is a terminator, it takes the terminator out of the teckel path.

If a Type 2 node N finds its right child is a terminator, it discards both its children and turns itself into a terminator. Any client waiting for N will find the terminator and see the set is exhausted.

If a client node finds its child is a terminator, it sees that the set is exhausted.

4.4 Run-time scenario, summarized

Type = G & status = NEW ⇒ It sets its status to `WORK` and detaches its left child, the 'generator body'. Next it makes a copy C of itself, attaches the generator body to C, and attaches C as its right child to itself, and clears its stop bit. Then it turns itself into a Type 2 node.

Type = G & status = WORK ⇒ The teckel detaches the generator body and splits it into a one-element 'head' and the rest, the 'tail'. Next it makes a copy of itself, attaches the head to itself, attaches the tail to the copy, and attaches the copy as its right child. Then it turns itself into a Type 3 node and is no longer a teckel stop.

Type = 3 & expression is finished ⇒ It alerts its parent.

Type = 3 & expression is *not* finished and its right child is a Type 3 node C whose expression *is* finished ⇒ N swaps its expression with that of C, removes C's stop bit, and makes itself a teckel stop. N can now alert its parent (an optimization) or wait for the teckel to come along the next time.˚ ˚ design decision

Type = 2 & whose child C has an expression that is finished ⇒ If the expression is `NO_RESULT`, N removes C by unlinking it from the chain, and is no longer a teckel stop.

If the expression is a value, it is compared to the values in V. If the test fails, C is removed, and N is no longer a teckel stop.

If the value is new, N adds it to its element list V, swaps V with the value of C, changes itself to a Type 1 node and its child C to a Type 2 node. The

Type 1 node is made a teckel stop, the Type 2 node is not.

Type = G & out of elements ⇒ It alerts its parent.

Type = 3 & its child C is an exhausted generator ⇒ It discards the left child of C and changes its type to "terminator".

Type = 2 & its child C is an exhausted generator ⇒ It discards C and V, and turns itself into a terminator.

Type = terminator ⇒ It alerts its parent.

Type = 3 & its right child is a terminator ⇒ It takes the terminator out of the teckel path.

Type = 2 & its right child is a terminator ⇒ It discards both its children and turns itself into a terminator.

4.5 Run-time scenario, summarized and sorted

Type = G & status = NEW ⇒ It sets its status to `WORK` and detaches its left child, the 'generator body'. Next it makes a copy C of itself, attaches the generator body to C, and attaches C as its right child to itself, and clears its stop bit. Then it turns itself into a Type 2 node.

Type = G & status = WORK ⇒ The teckel detaches the generator body and splits it into a one-element 'head' and the rest, the 'tail'. Next it makes a copy of itself, attaches the head to itself, attaches the tail to the copy, and attaches the copy as its right child. Then it turns itself into a Type 3 node and is no longer a teckel stop.

Type = G & out of elements ⇒ It alerts its parent.

Type = 2 & whose right child C has an expression that is finished ⇒

If the expression is `NO_RESULT`, N removes C by unlinking it from the chain, and is no longer a teckel stop.

If the expression is a value, it is compared to the values in V. If the test fails, C is removed, and N is no longer a teckel stop.

If the value is new, N adds it to its element list V, swaps V with the value of C, changes itself to a Type 1 node and its right child C to a Type 2 node. The Type 1 node is made a teckel stop, the Type 2 node is not.

Type = 2 & its right child C is an exhausted generator ⇒ It discards C and V, and turns itself into a terminator.

Type = 2 & its right child is a terminator ⇒ It discards both its children and turns itself into a terminator.

Type = 3 & expression is finished ⇒ It alerts its parent.

Type = 3 & expression is *not* finished and its right child is a Type 3 node C whose expression *is* finished ⇒ N swaps its expression with that of C, removes C's stop bit, and makes itself a teckel stop. N can now alert its parent (an optimization) or wait for the teckel to come along the next time.˚ ˚ design decision

Type = 3 & its right child C is an exhausted generator ⇒ It discards the left child of C and changes its type to "terminator".

Type = 3 & its right child is a terminator ⇒ It takes the terminator out of the teckel path.

Type = terminator ⇒ It alerts its parent.

5 Set and Sequence Termination

Several techniques are used in computer science to handle sequences of indeterminate length: NULL pointers, terminal nodes, counters, etc.

5.1 Possibilities

The following options have been considered for terminating sets and sequences of type T:

- The most obvious candidate for terminating a set or sequence is of course the NULL pointer.

  \+ It is simple and requires no special postscan action.

  - — It causes problems when the last expression in a set turns out to be `NO_RESULT`, and has to be eliminated from the set. Normally this is done by pulling-up of or overwriting by the next node. Overwriting can be done by Node itself, but pulling up involves changing a pointer in the Parent, and has to be done by the Parent, which must be alerted. But in this case there is no next node. So overwriting is not an option, and pulling up has the serious drawback that the Parent need not be and often is not of type T. So several node types besides T need to be prepared to do the pulling up; this is annoying.

- Add explicit terminators to some or all sets and sequences.
  - + There is always a node to use in overwriting, so pulling up can be avoided.
  - — It is hard to generate the terminators in the parser, so they have to be added either by the postscan or in the NEW sections of the reduce routines.
  - — Routines that need a value from a set or sequence now also have to test for a terminator.
  - — What are the `sort` and `data_type` of the terminator? `TERMINATOR` or `ADMIN` or ...? T? But it does not hold a T value. Whatever you do, the sequence or set is no longer homogeneous.
  - ☞ This is the present implementation.

- The main objection to using NULL as the terminator is the impossibility of adjusting pointers in the `Parent` node, but that does not have to be. It is in fact quite possible to determine the position of the pointer from which `Node` was reached. This pointer can be computed in C as

```
Parent == Root ?  &Root :
Node == Parent->child[0] ?  &Parent->child[0] :
Node == Parent->child[1] ?  &Parent->child[1]
```

  and can be used to update the `Parent` node, rather than alert it.

  When the teckel stops at the `NO_RESULT` node X, it finds its successor S (as its right child) and updates the pointer in the `Parent` node P to point to S; it does not alert P. The next time the teckel passes P, it is led to S (which may be the desired value or expression, or which may again be `NO_RESULT`).

  When X is next approached from another node P' one of the pointers in P' gets updated, and this process is repeated for all nodes pointing to X until X's reference count becomes 0 and X is freed.

  - + It avoids node overwriting.
  - + It allows the node itself to pull up the next node or NULL. It also allows, with a small extension, to have `Root==NULL` as the program termination condition, which is very satisfactory.
  - — It breaks the elegant simplicity of the very localized Teckel model.

  - — The `NO_RESULT` node(s) remain in the spine between the Type 1 nodes, and continue to slow down access to bona fide nodes.
  - —— It does not work. By pulling up NULL, one of the children of `Parent` P is set to NULL, but the administration of P does not know that. So the next time the teckel passes P it is sent into a blind wall! We could possibly program ourselves out of this: in addition to updating the pointer in P, we make it a teckel stop, so then next time the teckel stops there, it can analyse the situation. This would, however, affect many reduce routines, and would be a major kludge.

- Compromise: overwrite when there is a next node, but leave the `NO_RESULT` node as a terminator when there isn't.
  - \+ Hardly affects the rest of the program, since reduce routines have to check for `NO_RESULT` anyhow.
  - — Feels like a kludge.

5.2 Conclusion

There is not enough experience with sequences but the use of a terminator in sets is almost unavoidable. A set can consist of a number of expressions that all yield `NO_RESULT`. A program fragment requesting the next element of this set sends the teckel to a specific node. This node will after some time have to report that there will be no next element. To do this the node will have to be a real node, an "end" node of some sort; a NULL pointer cannot do the job and the other alternatives are unattractive.

6 Dynamic Function Identification

Since functions can be recursive they have to be identified at run time or a loop in the program tree would occur.

Suggestion: when asked upon to process a (named) function call the teckel calls a function with that name, which searches the `unit_sequence` to find the function, copies it and returns the copy. This avoids any loops in the program tree.

7 Dynamic Data Allocation

Dynamic data are those allocated through `Malloc()`. We want Teckel to be leak-free, so in order to check this property, we have to make sure that *all* mallocked data be returned at the end, including those that cannot threaten to cause the program to run out of memory. This requires disciplined memory management, which unsurprisingly has improved the program.

The major dynamic data structure in Teckel is the node. Its management is completely controlled by the reference-counting garbage collector. Pointers to nodes cannot be copied freely, and all actions have to go through the operations

```
struct node *new_node(void)
struct node *copy_tree(struct node *n)
```

```
void attach_tree(struct node **dest, struct node *src)
void overwrite_node(struct node *dest, struct node *src)
struct node *detach_tree(struct node **np)
void release_attached_tree(struct node **np)
void release_unattached_tree(struct node *n)
void release_terminal(struct node *n)
```

These preserve reference counts. See `node.[ch]`.

A node or tree is *attached* if it is reachable from `Root`. All routines that create a `struct node` pointer yield *unattached* trees. These trees must be either attached to the program tree or released before the end of the reduction, or `check_refcounts()` will fail. The routine `release_attached_tree()` is actually just a combination of `detach_tree()` and `release_unattached_tree()`.

The second in importance is the name list. Name list entries are created through various calls°, and freed at program termination by a call of `release_name_list(void)`.

° fill in

In third place we find the string. Mallocked strings originate as copies of fragments of the Teckel program text, identifiers, token names, etc. They are immutable, and pointers to them can be copied freely. They are created through calls of `permanent_string(const char *s, int l)`, which keeps a list of pointers to them. This allows the routine `release_permanent_strings(void)` to free all of them when the program ends.

The list is closed by minor dynamic data, for example the list used for keeping pointers during graph copying. These are freed as soon as their use has ended.

Proper use of these routines should result in zero memory loss at program termination. Memory usage and loss is reported by a call of `ReportMemoryLeaks(stdout)` from `Malloc.[ch]`.

8 Node Routine Entrance

The following considerations may simplify node routine entrance. A node routine is called when the node has been selected as a redex because `Node->is_teckel_stop` is on. So there is work to do in this node.

There are two kinds of nodes:

those which, in principle, needs all their operands to be finished; these nodes are 'strict';
the rest; these are usually for administrative purposes and are 'special'.

So if not all operands of a strict node are finished, the teckel sets `Node->flap_is_fixed`, so the other operand will continue to get processing power), unless the finished value was `NO_RESULT`, in which case that value can be processed depending on the nature of the operation of the node. ZZ this rambles

This testing and processing can be located in two macros to be used at node routine entrance: `is_strict()` and `is_special()`. They work as guards to make the processing more uniform.

9 Non-determinism

When non-determinism is encountered, the whole program (= process) must be copied; an implementation that shared the remaining common sections would make it very difficult to manage the (global) variables. This results in multiple copies and thus in multiple starting points for the teckel. When the teckel finds non-determinism, the system must have remembered which process it sent the teckel into, so it can copy that process.

The processes are kept in a circular list by the system, and a new copy is inserted at the back of the present process. This way all processes will get their turn. Note that this process is independent of the teckel redex-finding mechanism.

When a process yields `NO_RESULT` it is discarded.

Each process keeps a copy of its output, and each time something is added to this output, the whole output is printed. It is agreed that this is a kludge, to be corrected in Teckel 3.0, when a window will pop up for each process; if the process fails, the pop-up window disappears☺

10 Partial Evaluation

Partial evaluation of a function, on any of its parameters, is easy to implement. We pass the given parameter(s) as usual, and set the other parameters, which have the `Sort EXPRESSION`, to `DELAYED_EXPRESSION`. The `Sort DELAYED_EXPRESSION` propagates just like the `ValueSort NO_RESULT`. The partial evaluation stops when the top node of the expression (= function call) is `VALUE`, in which case we can harvest the value, or `DELAYED_EXPRESSION`, in which case a new function has been created.

When calling this new function, the `DELAYED_EXPRESSION` status must first be restored to `EXPRESSION`, to allow further evaluation.

11 Lazy Evaluation

Parameters are evaluated only when needed.

This causes a problem with memoization, which see (Section 14).

12 Eager Testing

This is just an example of eager testing.

In Section 3.3 we suggested a filtered set implementation by rewriting the expression to a sequence of $\{\mathcal{F}(x), x \in \mathcal{Z}\}$ commands, while the filter test was postponed to the last moment, when $F_0(a_1, ..., a_n)$ is computed. If we have partial evaluation (Section 10) we can try to perform the test earlier:

$$\mathcal{S} \equiv \{F_{n-1}(a_1) \textbf{ if } \mathcal{T}(a_1, , ...,), a_1 \in A_1\}$$
$$F_{n-1}(a_1) \equiv \{F_{n-2}(a_1, a_2) \textbf{ if } \mathcal{T}(a_1, a_2, , ...,), a_2 \in A_2\}$$
$$\cdots$$
$$F_1(a_1, ..., a_{n-1}) \equiv \{F_0(a_1, ..., a_n) \textbf{ if } \mathcal{T}(a_1, a_2, ...a_{n-1}, a_n), a_n \in A_n\}$$
$$F_0(a_1, ..., a_n) \equiv \mathcal{F}(a_1, ..., a_n)$$

where each call of $\mathcal{T}$, except the last one, is partially parametrized.

As soon as one test $\mathcal{T}(a_1, .., a_i, , ...,)$ yields *false*, all calls of $F_i(a_1, ..., a_1, , ..)$ are automatically suppressed.

13 Negative Results over Infinite Sets or Trees

Lazy evaluation and breadth-first search do not solve all problems: they don't provide negative results over infinite trees (= showing that an infinite tree does not contain a solution). Now it is true that such results cannot in general be obtained in finite time (Halting Problem), but we want to do what we can. Examples readily occur with filters:

$$6 \in \{x^2, x \in \mathbb{N}\}$$

which should yield `false`, but which, if no special measures are taken, will try $1 = 6$, $4 = 6$, $9 = 6$, $16 = 6$, ... and never see that it has missed the target.

Proofs of decidability of inclusion in an infinite set are usually based on producing the members of the set in a 'non-decreasing' order of some sort, for example based on their numeric values or the size of the members.

Note that we speak of "non-decreasing" rather than of "increasing". It is not necessary that subsequent members increase in value. It is not even necessary that they do not decrease: all that is needed is that there are identifiable members that will not be followed by smaller members. So actually we should speak of "eventually non-decreasing".

Perhaps we could try to implement something similar in Teckel; it could take the form of a bit with each set member saying "no subsequent member will be smaller/shorter than this one". Then, as soon as the 9 is received and $9 = 6$ fails because $9 > 6$, the result `false` could be delivered. This is difficult to implement° and may be addressed in Teckel 2.0.

° implementation problem

14 Memoization

Memoization is a great means for reducing time complexity from exponential to polynomial, and we would like very much to employ it. But it is far from clear whether it is compatible with lazy evaluation. Memoizing is at its best remembering function values given the arguments, but that situation does not occur with lazy evaluation: in general the arguments have not been reduced to values yet.

It may become necessary to try and determine the strict arguments of a function.

15 Resolved Issues

In no particular order.

- The teckel cannot make any modifications to the positions of its children, since other nodes may point to them. When a node turns into a value no new node is created; the old node is overwritten instead. This is necessary because more than one client may point to the node.

When we overwrite a node, it is likely that we we spoil the reference counts and node pointers in the overwritten node and possibly in the overwriting node. Since we actually only want to produce a node containing new information in the position of an existing node, the question arises which is preferable: overwriting the node and correcting the damage, or just inserting the new data into the old node.

Analysis shows that the only item that should not be inserted is the reference count, and that overwriting the child[] pointers involves in both cases releasing the old trees, and attaching the new trees, using the proper primitives. So overwriting, updating the reference count, and properly manipulating the child[ ] pointers is by far the simpler approach. It is implemented in `overwrite_node()`.

- The last statement when leaving a node must be one of

```
bye();                                   recalculate the teckel administration
Node->sort = DONE; alert_Parent();
overwrite_node(Node,x)
```

  Note that the pointer `Node` cannot be moved by the teckel, since this would invalidate the pointer `Parent`, and disallow any further calls of `bye()` or `alert_Parent()`.

- Arrays are implemented as sets of pairs, in which the system guarantees that the first members of all pairs are different (i.e., they are keys).

- The program should at all times be a dag to allow reference counting garbage collection.

- It would seem that constant definitions cannot be recursive, but this is not true. Counterexamples are the solution to the Hamming problem, and generative grammars.

- $x_1 \xi \ldots \xi x_5$ is shorthand for $x_1 \xi x_2 \xi x_3 \xi x_4 \xi x_5$, For a relational operator (an operator with type $(X, X) \rightarrow \mathit{Boolean}$) this converts to $x_1 \xi x_2 \wedge x_2 \xi x_3 \wedge x_3 \xi x_4 \wedge x_4 \xi x_5$.

- Roll-out of sets as actual parameters corresponding to single formal parameters. The call

  $$F(a_1, ..., a_{k-1}, S_e, p_{k+1}, ..., p_n)$$

  is transformed into

  $$\{F(a_1, ..., a_{k-1}, e, p_{k+1}, ..., p_n), e \in S\}$$

  where $a_{...}$ are type-wise fitting parameters; S_e is a set of e and the k-th parameter of F is of type e; and $p_{...}$ are arbitrary parameters. Repeat for all mismatches.

  This also works for '+', '×', etc.

- Optimistic graph visiting – When visiting the nodes of the program graph, nodes must be marked to control the depth-first visiting process and preserve linearity. To avoid the bother of cleaning the markers before or after the scan, a global counter `scan_count` is kept, which is increased for each scan, and which is used as the marker. If the marker in a node is equal to the present `scan_count`, that node has already been visited during the present scan.

  This set-up leads to very simple code. Since increasing an integer in C has a cycle of 2^{32}, this optimistic scheme only fails if a node is not visited for $2^{32} - 1$ scans (about 4 billion) and then visited on exactly the 2^{32}-th scan.

  A disadvantage of this scheme is that it makes graph visits non-reentrant: no graph visit van be started from within a graph visit.

- Although one would expect many of the nodes in the program to bear syntactic information only, irrelevant at run time, 110 of the 118 syntactic node types survive to run time.

- Initially all nodes are teckel stops. This forces us to pay attention to all node types at least once, and tends to prevent the teckel from running into a NIL pointer.

16 Current Issues

- There is a relation between the number of children with !DONE status (the active subtasks) and and being a teckel stop, but the relation is murky.

  Values always have DONE status. All nodes with DONE status are teckel stops. But what about nodes with DONE status that are not values? Values have no subtasks but is having no subtasks the same as being a teckel stop?

  The cause of this confusion is that the child pointers in Teckel node serve multiple purposes:

  - The basic Teckel model is composition-based: the program is a dag that must be reduced to a value; and each node has 1 or 2 children, from the value of which the value of the node is derived. In this model the number of active subtasks is meaningful, and that number being zero correctly signals a teckel stop, so the node value can be computed. The status is then set to DONE.

    The value of subtasks of a node is computed from the !DONE (i.e. NEW or WORK) status of its children.

    → All pointers to !DONE children are teckel paths. They all count as subtasks.

  - Some nodes have !DONE children that are temporarily blocked from being processed. One example is the **if** (**then**, **else**) node, which first has to compute the if-part.

    → Some pointers to !DONE children are temporarily blocked teckel paths. These do not count as subtasks.

– Some pointers to children are not teckel paths at all, but just ways to connect data to the program tree. Examples are:

1. pointers to data attached to a generator node, required for generation; in this case the children are not accessible to the teckel, and their status is immaterial.
2. the right child of a Type 1 node, where the path is just the pointer to the next element; in this case the child's status is unclear.

→ Some pointers to children are just pointers to data, not teckel paths. These do not count as subtasks.

The above conclusions (marked →) are contradictory. Two additional statuses (LIMBO and DATA?) could help, but it feels like overkill. Also, DATA is not really a Status since it cannot change; it is more a Sort.

- Short coding for post-flip: position 2: `is_teckel_stop`; pos. 1: need-to-flip; pos. 0: direction. Elegant. But useful?
- `Node->status = DONE;` and `alert_Parent()` always occur together. Combine?

Appendix

A Weak Pointers

[Weak pointers were tried in the implementation of sets, to solve the problem of duplicate values. The idea was to keep the finished values in the spine where they originated, so they were available for comparison. This required the top of the spine to remain accessible, even when no process was interested in its first element any more. Keeping a pointer to it in the other spine elements would cause cycles in the tree, and cause the reference-counting garbage collector to fail, so a weak pointer was generated in the form of a unique name, which was stored in the name list and made to refer to the spine top node.
The number of copies of the name handed out was counted in the name entry, and when it reached zero, the name entry was released, and with it the pointer to the spine top it contained.
Problems arose, however, when parts of the program tree had to be copied, and nodes had to be analysed and checked for the presence of such names, so their name reference counts could be updated. Although there is in principle no reason why it should not be possible to do this correctly, it soon got messy and clumsy, and began to look like a second reference-counting garbage collector.
Conclusion: Weak pointers may not always be a good idea.]

B C Module structure

A module `M` in C consists of an interface file `M.h`, to be presented to the user, and a code file `M.c`, to be compiled separately. The compilation of the code file may require additional parameters, like preferred buffer size, default file names, etc. These parameters must be supplied by the user. They should not be in `M.h`, which presents information *to* the user, but they should not be in `M.c` either, since that file is outside the user's jurisdiction.
In principle such parameters are instantiation parameters. They should be in a specification file `M.spec`; the code should be in a definition file `M.body`; and there should be a program to substitute the specification parameters from `M.spec` in `M.body`, producing an instantiated `M.c`.
As long as we need only one instance of a module the full mechanism is not needed in C: instantiation can be done by C's macro processor.