Teckel Design Considerations

Dick Grune
dick@dickgrune.com

October 13, 2014; DRAFT

1 Resolved Issues

In no particular order:

- All global names must be unique. All local names must be unique within their locales. There is no identifier hiding.
- Parameter matching, as in the call $F(a, b, a, b \in V)$ results in a set; it is an abbreviation of $\forall\ \{a \in V\} \forall\ \{b \in V\} F(a, b)$.
- There is no array denotation: arrays can be filled only by assignments.
- Numbers are represented as rationals with a numerator and a denominator. This is required to make expressions like $\frac{1}{2}n(n-1)$ work.
- The value `NO_RESULT` does not propagate to sets. We want to show the successive values in a set and cannot wait to see if `NO_RESULT` ever pops up.
- We do not accept multiple identifiers in a numerical range `typed_definition`: `1 < i, j <= 10` looks too much like two comparisons.
- We do not do any source code optimization: `x - x` is not optimized to `0`. The reason is that the computation of `x` may not terminate or may yield `NO_RESULT`; in that case `x - x` will not terminate either or will yield `NO_RESULT`. Pathological code begets pathological results.
- The `{}` inside a comment have to match and nest, to allow the inclusion of Teckel code. This means that comments nest automatically, even though a `\Tcomment` token inside a comment is not recognized by Teckel; but its `{}` pair is. Note, however, that LaTeX will recognize any LaTeX command inside a comment.
- We don't want $a/b \times c$, since science interprets this as $(a/b) \times c$, and schools interpret this as $a/(b \times c)$. And $a/b/c$ raises eyebrows too. So we accept one division after a number of multiplications. Example sequences are:

  $a \times b \times c \times d/e$
  $abcd/e$

- Although conditional expressions are probably rare in abstract algorithms, we need them; see, for example Section "Filtering" in the Implementation Design. A notation with `\Tif` etc. would be confusing, so we resort to the C construction `bool ? expr1 : expr2`. We also want `bool ? expr`, which yields `NO_RESULT` if `bool` fails. If parsed as `((bool ? expr1) : expr2)`, the conditional expression contains two binary operators, each interesting in its own right:
  - `bool ? expr`: yields `expr` if `bool` evaluates to `True`, and to `NO_RESULT` otherwise.
  - `expr1 ? expr2`: yields `expr1` if it exists (= is not `NO_RESULT`) and `expr2` otherwise.
- Occasionally Teckel assumes it gets correct LaTeX code. For example, LaTeX does not allow `a^b^c`, but Teckel does not implement this restriction.
- Functions are not values; Teckel is not a functional language. So there is no $\circ$ operator to compose functions.
- Although it would be in good Teckel style to process multiple output commands breadth-first (and actually easier to implement), that feature probably has no added value, and results would appear interleaved.

  Resolution: We allow multiple output commands, but execute them sequentially.
- There is only one source of non-determinism: the if-statement without else-part. This if-statement can be explicit, as in '**if** $a > 0$ **then** a', or implicit, in the use of multiple function declarations. Each declaration of a multiple function may be guarded by an if-statement or by a parameter pattern. If several declarations match, several new programs result; if no declaration matches, the value `NO_RESULT` results, which is passed upwards by further teckel actions. [This requires some handstands when providing the non-deterministically intermixing output.]
- Is `F(x, y)` a function of two variables, `x` and `y`, or of one compound parameter, `(x, y)`?
  1. It makes `F` an operator working on a pair.
  2. It allows functions `F(x, y) -> (x, y)` to be employed in transitive closure, which seems useful.
  3. How about partial parametrization?

  Resolution:

  Ad 1. It would also allow a call `F p` where `p` is a pair of the right type; this would be artificial.

  Ad 2. This can, and should, be achieved by `F((x, y)) -> (x, y)`.

  Ad 3. Indeed.

  Conclusion: If `F(x, y)` is a function of one compound parameter `(x, y)`, it should be declared as one: `F((x, y))`.

- `\Toutput` commands should be executed sequentially. Breadth-first output is very confusing and seldom useful; perhaps a command-line parameter.

2 Current Issues

In no particular order:

- Global variables must be initialized at the global level, to allow identification. Global constants cannot depend on variables. Because global constant definitions may call functions, this requires checking for functions that use (global) variables; so there are "constant functions" (pure functions) and "variable functions".˚ ˚ postscan
- The semicolon operator belongs to the world of imperative programming, and does not allow lazy evaluation. It evaluates all variables available at that point.
- Do we have strings? Are they just a notation for sequences of characters? Operations on them? Currently `"` is just a `large_identifier`; `\"` is illegal. [Just check the code for `'"'`.]

  Do we have characters? For reporting only? The `'` is used as a prime: `a'`; `\'` is again illegal.
- Sequences are almost always sequences of atoms. Is it usseful to restrict sequences to atoms? What about `1..3`?
- We should not be using LaTeX braces (`{}`) for creating a syntactic structure that is not visible on paper. Exception: we accept `_{i}`, since we have to accept `_{i+1}` since in the latter the effect of the `{}` show on paper. The same applies to `^{i}`. Are there more exceptions like this?
- ˚ Is ρ^∞, which is $\rho\rho\rho\cdots$, always equal to ρ^*, which is $\rho \cup \rho^2 \cup \rho^3 \cdots$? No, $\rho^* \neq \rho^\infty$. More in particular, if ρ is viewed as a graph, $a\rho^* b$ is true if b can be reached from a; $a\rho^\infty b$ is true if b can still be reached from a after arbitrarily many steps, i.e., when b is on a cycle reachable from a. [1]It is doubtful if there is any use for ρ^∞. ˚ get a good definition of transitive closure
- Scanning an array: looks like we need a notation for the domain of an array. ˚ Alternatives: Awk uses `for` (i `in` a) *body* to scan the array a, but the i is still the index, which is weird since it is not in a. Python uses built-in iterators to do this, which does not fit the Teckel model. ˚ domain of an array
- Remarkable: The transitive-closure feature allows recursive functions without an escape hatch:˚ ˚ get syntax

  ```
  MAX(a, b) \equiv MAX(a, a) \mathbf(if) a >= b
  MAX(a, b) \equiv MAX(b, b) \mathbf(if) b >= a
  ```

  Now calling $MAX^*(2, 3)$ will converge on MAX(3, 3).

  Is there a way to catch this and yield the 3? And should we?

[1] So ρ^∞ is a subset of ρ^*. Is there an interesting form for $\rho^* \setminus \rho^\infty$?

- Crazy idea: Teckel has $\mathbb{N}$ and $\mathbb{Z}$ for natural and non-negative numbers, resp., but the rational numbers are countable too: produce all rational numbers N/D with $N+D = S$ for increasing S and throw out any rational in which N and D have a common factor. Suggested symbol $\mathbb{Q}$.

3 Result, absent, and error

˚ ˚

The evaluation of an expression results in a value, which can, in principle, be a result, absent, or an error. terminology

3.1 Combining absent and error into `NO_RESULT`

An absent value results from the failure of an if-condition on a value, or from the arguments of a call not matching any definition (which is actually again if-condition failure); there just is no value to return. If it ever reaches the top, nothing is displayed, since that is the answer.

An error results from a faulty computation, and carries with it an explaining error message; if it ever reaches the top, it is displayed.

Although it is easy to think of instances of 'absent', abstract algorithms have few opportunities to cause a 'faulty computation'. The following possibilities come to mind:

- A reference to the value of a non-existing array element. This can be considered as a case of `NO_RESULT`, which would actually fit in well with the semantics of

  $$i < 10 \wedge a_i \neq 0$$

  when $i = 10$: it is clearly the intention that a_{10} should not be referenced, but in breadth-first evaluation it might be nevertheless. (An *assignment* to a non-existing array element just creates the element.)
- Division by 0. But that yields $\pm\infty$ (non-deterministically?).
- The expression $0/0$.˚ ˚ To do
- End of input. But Teckel programs do not read input.
- Integer overflow.˚ ˚ To do
- The use of an unassigned variable. The same argument as for the non-existing array element applies in many cases, but it might also be a genuine error.

So we compromise, and produce the –slightly contradictory– value `NO_RESULT` with an explaining string attached; this string may be an error message or `"absent"`. Further operations will then have to decide what to do.

3.2 The propagation of NO_RESULT

The propagation of result and NO_RESULT depends on the operator they are involved in.

Monadic operators pass the value straight on.

Dyadic operators that "normally" require both operands (the 'strict' ones) yield NO_RESULT if any is NO_RESULT and the result otherwise. This implies that if any operand is NO_RESULT we can chuck the computation of the other operand, and immediately yield the value NO_RESULT.

The situation with the lazy Boolean operators $\wedge$ and $\vee$ is more complicated. As explained above, the idea of a symmetric lazy $\wedge$ is that if one operand is *false* the other should not be computed. This means that $\wedge$ should wait for the other operand if the first (not necessarily leftmost) operand yields NO_RESULT or *true*, and then decide according to the following table:

	false	NO_RESULT	*true*
false	*false*	*false*	*false*
NO_RESULT	*false*	NO_RESULT	NO_RESULT
true	*false*	NO_RESULT	*true*

The table shows that we can abandon the computation as soon as one operand turns out to be *false*, which fits in with the classic logic rule $\mathit{false} \wedge ... = \mathit{false}$.

Likewise, the the idea of a symmetric lazy $\vee$ is that if one operand is *true* the other need not be computed, although the argument is weaker here since the text reads 'need not' rather than 'should not'. If the first available operand value of $\vee$ is *false* or NO_RESULT it has to wait for the other operand; otherwise it should just yield *true*.

When NO_RESULT is added to a set, the NO_RESULT disappears. This means we do not need to retract a set as the top-level result when we have already printed elements of it, and that expressions like $\{x \in \{3,5\} \mid x = 2+2\}$ yields the empty set as it should, since it is interpreted as $\{3 \textbf{ if } 3 = 2+2, 5 \textbf{ if } 5 = 2+2\} = \{\texttt{NO\_RESULT}, \texttt{NO\_RESULT}\} = \{\}$.

When NO_RESULT is added to a sequence, the whole sequence becomes NO_RESULT; after all, NO_RESULT is not the empty sequence.

When NO_RESULT comes up as a value in a struct, the whole struct becomes NO_RESULT.

User operators are defined by Teckel code, to which the above rules apply.

4 Formals, destinationals and actuals – lvalues and rvalues

Formal, destinational (=lvalue) and actual (=rvalue) parameters are similar in that all three are concerned with the same data types. These are

* basic types: atoms, Booleans, rationals
* structured values: structs (=records), sequences (=lists)
* unstructured values: sets

Formal and destinational parameters are very similar in that both specify a set of locations to put the actual values in.

The basic receptacle of a value is the identifier, or, in the case of a destinational parameter, an array element or array segment. The receptacle is formal in a definition and global or local in an assignment.

There is only one kind of actual parameter, the expression.

4.1 Destructuring

Structured values can be destructured in a parameter transfer or assignment; this requires a special syntactic form for the receptacle. Traditionally this form is chosen to mimic the syntactic form of the structured value as an actual parameter, and we follow that tradition. Example:

$$(a, b) := x$$

Structs (records, multiple values) are usually represented by enclosing them in `()`. An actual parameter could be `(101, 80, 70)`, and the formal/destinational destructuring syntax `(length, width, height)` corresponds closely to this. This form is fixed because the number of elements in a struct is fixed.

Sequences (lists) are created basically by concatenation: *abc* is a sequence of 3 elements, each of which may be a single element or a sequence. Destructuring takes place by matching the result with a similar lvalue expression: `p q r`. The matching process is controlled by the types of the lvalue identifiers. These can be constants of type T, variables of type T and variables of type T^* or type T^+; type definitions can be attached to the receiving parameters.

A constant formal/destinational must match the actual value. A variable of type T must match one actual value. A variable of type T^* (T^+) must match any (non-zero) number of actual values.

Sequences can be created using several syntactic forms, but not all are suitable as formal/destinational forms.

Integer sequences can be created using ranges: $i \ldots j$. Using a similar form as a formal or destinational, for example $p \ldots q$, would discard the contents of the sequence and just set p to i and q to j. Such a formal or destinational parameter would be confusing and pointless.

Sequences can also arise as subsequences from previously created sequences. There are four notations for this:

$S_i \ldots S_{i+k}$
$S_{i \ldots i+k}$
$S[i] \ldots S[i+k]$
$S[i \ldots i+k]$

(perhaps in more than one dimension). The formal/destinational forms match these, but the matching is not obvious since the actual value supplies length and contents, but not the bounds. In formals the lower bound has to be a constant or an (integral) number; the upper bound must be a variable, which is then set from the length of the actual. The result is a sequence with the indicated bounds. The same applies to destinationals, except that the lower bound may be an (integral) expression. The assignment replaces the indicated elements from the sequence by those form the actual value, and sets the upper bound variable.

Subsequences can be single-length; they are then just elements. The notation for the actual value is S_i or $S[i]$. This notation is immediately usable for destinationals. For formals it seems to denote the creation of a subsequence of length 1, which would be pretty useless.˚ ˚ no single-length formal subsequences

Sets cannot destructured and can be received in an identifier only.

Formal parameters can conventionally be supplied with type definitions, to aid in reading and to lend support to the type analysis. Subsequence matching parameters must be supplied with type definitions, to distinguish between single-length and multi-length matchers.

A list of formals or destinationals first lists all the parameters and then the (optional) type definitions for the lot. But how about $F(a \in U, b, c \in V)$.˚ ˚ problem

In total we obtain

type	formal	destinational	actual
value	idf[, types]	idf	expression
subsequence	idf idf ..., types	idf idf ..., types	oper ...
struct	(formal, ...)	(dest, ...)	(expr, ...)
sequence	$x[1..k]$[, types]	$x[i..j]$	$x[i..j-1]x[j+1..k]$
integer seq.	$x[1..k]$[, types]	$x[i..j]$	$1 \leq i \leq 10$
element	–	$x[i]$	$x[i]$
set	idf[, types]	idf	{ expr, ... }

5 Non-determinism

In most cases, non-determinism[2] arises from multiple function or operator definitions, for example

$$a \pm b \equiv a + b$$
$$a \pm b \equiv a - b$$

So 5 ± 4 yields both 9 and 1. Something similar happens with grammar rules:

$$S \equiv x$$
$$S \equiv x\ S\ x$$

Below we will use the trivial example

$$F \equiv 3$$
$$F \equiv 5$$

(which by the way shows that there are non-deterministic constants).

5.1 Two interpretations of non-determinism

There are two interpretations of the notion of "non-deterministic":

- Floyd's ("Nondeterministic Algorithms", 1967): all allowed values of a non-deterministic expression are tried, and those that do not end in failure are the results. If there are zero allowed values, that's OK: then there are zero results. So a non-deterministic program may have zero, one, or more results.

[2]The Oxford Dictionary has "non-determinism" rather than "nondeterminism".

- Dijkstra's ("Guarded Commands, Nondeterminacy and Formal Derivation of Programs", 1975): out of the allowed values of a non-deterministic expression one is chosen, and the computation continue with that value. When there are zero allowed values, the program is in error. When another choice would have yielded another value, that value is also a possible result of the program; but the underlying assumption is that if there is more than one choice, they should all lead to the same answer, or the program is kind of iffy (not Dijkstra's term☺).

Dijkstra's non-determinism is a very tamed non-determinism, almost making the program deterministic. Teckel uses Floyd-style non-determinism.

When a multi-defined function is called, multiple programs result, each with one of the definitions. This sounds reasonable, but has remarkable consequences, which should be considered.

5.2 Independent Computation

If we compute $F = 3$ we get two answers, *true* and *false*, which is OK. If, however, we ask $F = 3 \wedge F = 5$ we get the answer *false* (three times) plus the answer *true*:

$3 = 3 \wedge 3 = 5 \rightarrow \mathit{false}$
$3 = 3 \wedge 5 = 5 \rightarrow \mathit{true}$
$5 = 3 \wedge 3 = 5 \rightarrow \mathit{false}$
$5 = 3 \wedge 5 = 5 \rightarrow \mathit{false}$

This could be avoided by evaluating F just once, but

1. that would be unnatural: when it's written twice it's evaluated twice; for example, the Teckel program

   $P \equiv a$
   $P \equiv b$
   PP

   should yield four results: `aa`, `ab`, `ba`, `bb`;
2. it would be difficult to arrange this for functions with several, perhaps complicated arguments.

The multiple definition of F above was unconditional, but often such definitions come with a condition, a 'guard':˚

˚ syntax not yet decided

abs $a \equiv +a$ **if** $a \geq 0$
abs $a \equiv -a$ **if** $a \leq 0$

in which we use both $\geq$ and $\leq$ to keep the symmetry of the operation. But the unconditional definitions above can easily be cast in the same format:

$F \equiv 3$, **if** *true*
$F \equiv 5$, **if** *true*

This suggests that the basic source of non-determinacy is the if-statement without an else-part. In other words, **if** *condition* **then** *expression* without an **else**-part is shorthand for **if** *condition* **then** *expression* **else** `NO_RESULT`.

5.3 Forcing non-determinism

The expression

$$\exists x \in \{3, 5\} \mid x = F$$

produces two copies of the program, both continuing with *true*; and

$$\exists x \in \{3\} \mid x = F$$

continues with one *true*, for $F = 3$, and one with one *false*, with $F = 5$. Given the way non-deterministic algorithms use the non-determinism, one might suggest that in the latter case the program should have been abandoned.

Considering the test in the filter to actually be an if-statement without an else-part is not the solution:

$$\exists x \in \{3, 5\} \mid x = 4$$

should just yield *false*, and

$$\{x \in \{3, 5\} \mid x = 4\}$$

should yield the empty set.

The desired effect (just yielding the single answer *true*) can be achieved by

$$\exists x \in \{3\} \mid \textbf{if } x = F \textbf{ then } \mathit{true}$$

which makes explicit that non-determinism is required.

5.4 Domain Failure

Another phenomenon occurs when we call a partial function with an argument outside its domain. Given the definitions

$$F(3) \equiv 3$$
$$F(5) \equiv 5$$

which are equivalent to

$$F(n) \equiv 3, \textbf{ if } n = 3 \textbf{ then } \mathit{true}$$
$$F(n) \equiv 5, \textbf{ if } n = 5 \textbf{ then } \mathit{true}$$

a call of $F(4)$ yields `NO_RESULT`, which may eventually cause the program to be abandoned. The same occurs dynamically in

$$\mathrm{Fac}(0) \equiv 0$$
$$\mathrm{Fac}(n) \equiv n \times \mathrm{Fac}(n-1), \textbf{if } n > 0$$

and the call $\mathrm{Fac}(-1)$.

It should also be noted that $0 < F < 10$ is not the same as $0 < F \wedge F < 10$ if F is non-deterministic. Teckel assumes all expressions to be non-deterministic, until it has learned to determine otherwise (see Section 5.2).˚

˚ postscan 2.0

5.5 SETL's OK oracle

Under Floyd's interpretation of non-determinism the SETL oracle OK is trivial to implement:

OK ≡ **true**

OK ≡ **false**

5.6 Non-deterministic variables

There are no multiple-definition (non-deterministic) variables.

6 Over-all Structure

The Teckel project consists of three phases: lexical analysis; prescan & parsing & postscan; and the run-time system. See Figure 1

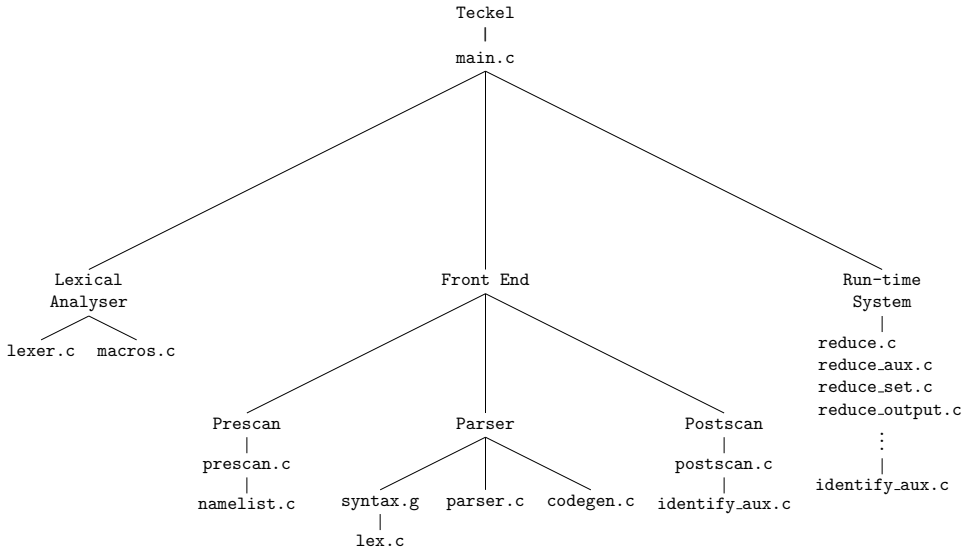

Figure 1: Over-all structure of the Teckel project