Postscan Design Considerations

Dick Grune
dick@dickgrune.com

October 26, 2014; DRAFT

1 Introduction

The postscan has two distinct purposes: 1. to do context checking; and 2. to make life easier for the interpreter, by presenting it with a simpler program tree than the syntax and the parser can provide.

Context handling consists of the traditional parts identifier identification and type determination and checking (Section 4). Function identification is special (Section 6).

Tinkering with the program tree is occasionally useful and sometimes necessary, but it has its price: the program tree no longer conforms exactly to the syntax, and that can be confusing. The problem is exacerbated by the fact that many transformations could in principle be done equally well on the fly by the teckel as well as statically by the postscan. To avoid a grey area, the postscan should only do two kinds of restructuring:

1. trivial changes which turn special cases in the syntax into normal cases;
2. major changes which cannot reasonably be done by the teckel.

The first kind should not impact the interpreter. An example is replacing the node type `R_expression_list_proper` by `R_expression_list`.[1]

The second kind should be so big that no confusion can occur. Prime example is the development of filtered sets into conditional expressions, functions and simpler sets

Other tasks for the postscan have been considered (see Current (Section 11) and Resolved (Section 12) Issues), but none have been found necessary to date.

2 Simplifying small identifier and large identifier

The distinction between `small_identifier` and `large_identifier` is made by the lexical analyser and is meaningful to the parser only. It is just annoying to the postscan and the interpreter, so the first thing the postscan does is replacing them both by `identifier`.

[1] The syntax rule `expression_list_proper` is an anomaly in the grammar, to avoid the form `(a)` from being ambiguous: a packed expression or a tuple of one element.

3 Node Properties

Presently it is awfully complicated.
A node has the following properties:

Property	Example(s)	Motivation
node_type	large_identifier, R_program, ...	indicates its function in the parse tree
sort	IDENTIFIER, EXPRESSION, VALUE, ...	determines additional fields
class	CONSTANT, VARIABLE, FUNCTION	???
scope	GLOBAL, FORMAL, LOCAL	determines identification terrain
data_type	NUMERIC, ATOM, ...	determines applicable operations
(RT) status	NEW, WORK, DONE	tallies progress during RT

We have the following identifier node_types, all informative; information determined by the prescan (what information??):

identifier (combined from small_identifier and large_identifier)
prinfix_operator_token
postfix_operator_token
function_identifier
constant_identifier (not in parser input)

The following groups of nodes need a data_type:

1. sort = EXPRESSION
2. the above identifiers → sort=IDENTIFIER

Only identifiers need scopes.
Factoids:
Identifiers are not expressions:

1. determined by their strings
2. have scope

Expressions are not identifiers:

1. determined by their children

Does atom ρ have sort=IDENTIFIER or sort=EXPRESSION?
needs computation:

1. sort=EXPRESSION

needs no computation:

1. sort=VALUE (incl. atom)
2. sort=CONSTANT

ATOM is a data_type and its values are identifiers ⇒ an identifier can have as its value another identifier CONSTANT? atom

4 Name identification

A name has GLOBAL, FORMAL or LOCAL scope.

All global names must be unique. All formal and local names must be unique within their locales. There is no identifier hiding, so no formal or local can be identical to a global name. As an exception° to this we allow formals to be identical to global constants: people write things like: "declare F(a,b), and now try this for a=1 and b=0". ° implement

A name refers to an object of a basic type, VOID, ATOM, BOOLEAN, and NUMERIC, or of a compound type, SET, SEQUENCE, ARRAY, or TUPLE. In compound types, the elements of the object carry indications of their (sub)types.

A name identifies an object of class CONSTANT, VARIABLE, or FUNCTION.

So we can have a GLOBAL BOOLEAN CONSTANT, a LOCAL NUMERIC VARIABLE, etc.

Type combines orthogonally with both other dimensions, but not all combinations of scope and class are meaningful:

		class		
		CONSTANT	VARIABLE	FUNCTION
scope	GLOBAL	✔	✔	✔
	FORMAL	✔	✘	✔
	LOCAL	✔	✔	✔

Scope is shown syntactically, from the position in the program tree: outside functions, inside formal parameter structures, and inside a function but not formal. Class is shown lexically, from the use of \Tdefine, \Tassign, or \Tfunction; compound type derives from the syntax, from forms like set_pack, expression_element_list, etc.;° and the basic types BOOLEAN and NUMERIC derive directly from the lexical item. Atoms are GLOBAL CONSTANTs. ° incomplete

But identifiers are used for multiple purposes: atoms, constants, variables, formals, functions, and user operators. So we need an algorithm to identify atom, constant, variable, formal, function, and user operator names.

In more detail:

- Recognize global constants (in global \Tdefine{identifier}), global variables (in global \Tassign{identifier,...}), and functions (which are always global, even when defined locally).
- Recognize local constants (in local \Tdefine{identifier}) and local variables (in local \Tassign{identifier,...}), which must be disjunct and different from the global names.
- Any name not recognized so far, used in a global constant or variable definition or in the output expression is an atom.
- Any name not recognized so far, used in a function body and not occurring in its parameter list is an atom.
- Any name not recognized so far and occurring in the parameter list of a function is a parameter to that function.
- This process leaves no name unrecognized.

How about atoms in formal parameters, for matching purposes?° ° problem

Names can be:

atoms	global
constants	global/local
variables	global/local
prinfix	global
postfix	global
function	global
formal	local

5 Data Type Identification

In its simplest form data type identification is a bottom-up process: informative tokens have intrinsic data types, they are operand to operators which then get their data types, and so on. This is no longer true when an identifier is met. The postscan teckel will then have to locate the definition, but it may not have visited that definition yet, and there may be no data type available there. So transitive closure is needed.

This mechanism may not work for recursive entities and will probably not work in the presence of type inference.

6 Function and Operator Identification

Functions and operators can only be defined on the top level. They are characterized by a profile consisting of their name and three numbers: the number of left operands (0 or 1); the number of indexes; and the number of right operands / parameters.

Functions and operators can be recursive, so they cannot be identified and linked in like formals and locals by the postscan, because that would cause loops in the program graph. Even zeroadic operators can be recursive:

$$N \equiv \{1\} \cup \{n + 1, n \in N\}$$

So identification has to be performed at run time, when a name is met. It is then that its definition must be found. The postscan can make sure that indeed such a definition can be found.

7 Set Generators

The various forms of set generators are so different that they all have to be clobbered into the form
with $F(x) \equiv \mathcal{F}(x)$ **if** $\mathcal{T}(x)$, where $\mathcal{F}$ is the function that yields the elements and $\mathcal{T}$ is the test in the filter.

8 The Invisible Operator

The Invisible Operator can be

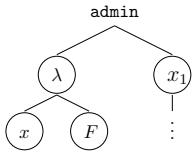

Figure 1: Set generator data structure

- the multiplication operation on numerals: only the first primary can be a number: `2x` is acceptable, `x2` is not. Rule: `number`$^?$ `< idf`* `< (pack or |E|)`*.
- the concatenation operator on ATOM sequences and ATOM singletons, where we allow any combination.

The postscan can use the data type to distinguish between the above; the rule can be implemented using a FS automaton on the node type.

9 Relation to the Syntax

The postscan relies on certain properties of the syntax.

- `Boolean_expression_list`s occur in `filters` only.
-

10 Relation to the Run-time System

The purpose of a Teckel run is to produce and display a value. Nodes that carry a value need a data type and the postscan needs to provide these. To this end the postscan has a list of node types that need data types in the routine `needs_data_type()`, extracted by hand from `syntax.g`. For each of these node types there must be code to determine its data type in `do_type_node()`. (Until we introduce type inference, a repeated bottom-up process suffices.)

Since it is awkward (though possible) to merge these two lists, there is a shell script, `check_node_types`, which checks the identity of the two lists.

The run-time system has code to reduce each of the value node types, in `reduce()`. But `reduce()` also reduces other node types, f.e. `R_statement_sequence` or `R_program`. The shell script `check_node_types` also reports about the code in `reduce()`.

11 Current Issues

- Parameter lists
- Expressions with Ellipses

12 Resolved Issues

- The postscan modifies the tree to produce structures that the parser cannot provide, but in many cases the teckel could also produce those changes at run time. So where is the boundary between the postscan and the interpreter? One extreme is to do all static computation in the postscan: converting the numeric constant `"3"` to the value `3`; restructuring the function definitions; etc. This soon looks like precomputation (optimization). The other extreme is to do minimal postprocessing.

  Some thoughts:

  - The code for a job in the postscan is similar to that in the interpreter.
  - Postscan and teckel run are not fully equivalent:

    The postscan cannot set local variables in the nodes since they do not exist yet at postprocessing time.

    The teckel cannot modify parts of the tree it never reaches (administration code, etc.), although it usually visits a node from which such pars can be reached.
  - Touch-up to combine two similar syntactic constructs is OK.
  - Major restructuring to implement complicated semantics in terms of existing features is OK.

  Conclusion: see Introduction (Section 1).

- Due to the complicated syntax around the `^` operator, the operator occurs in a syntax setting that differs from that of `+`, `-`, `\times`, `/`, etc. Corrected by patching the parse tree.

- There are two kinds of generated sets, type-based, as in `\{ a^2, a \in \mathbb{N} | a \mod 2 = 0 \}`, and range-based, as in `\{ 0 < i <= 10 \}`. The full form of a type-based set specifies the types (and with them the generators) of all bound variables in the expression to be generated:[2]

  ```
  \{( a, b, c ), a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N} ~|~
      a^2 + b^2 = c^2 \}
  ```

  but for convenience this may be condensed to

  ```
  \{( a, b, c ), a, b, c \in \mathbb{N} ~|~ a^2 + b^2 = c^2 \}
  ```

  The special case in which the expression consists of just one variable

  ```
  \{a, a \in \mathbb{N} ~|~ a \mod 2 = 0 \}
  ```

  can be abbreviated to

  ```
  \{a \in \mathbb{N} ~|~ a \mod 2 = 0 \}
  ```

[2] Note that the three `\mathbb{N}`s in the example are three independent generators.

The postscan can easily reconstruct the full form in both cases.

Note that it is the `\in` separator (which is not a Boolean operator here!) rather than the (optional) filter separator `|` that identifies the set expression as a type-based one. If the `\in` is missing, the filter has to be absent too, and the set expression is an enumerated set, or a syntax error will occur.

- The postscan has to make corrections to the program tree, and has to do so repeatedly, since some modifications may result in constructions that require further modification. But sometimes it replaces a construction by an almost equal construction, and then it is difficult to see if the modification has already been done.

  It would be possible to separate the modifications in "single-shot" and "repeated", but that would probably be error-prone. No part of the node is available and initialized to put a marker in. So a call of `Transitive_Closure(Root, init_node_info0)` can be used to initialize the `info[0]` field of every node to 0 so it can be reliably used a marker by the rest of the postscan.