SAS-FP2 Specware Specification and Refinement with Function Composition

1. Formal Specware Specification of the Function Composition Engine

```specware
spec FunctionCompositionEngine is
  import /Library/Base

  % Types and Sorts
  type SASStatement = String
  type DataStep = {name: String, body: List String}
  type ProcStep = {procName: String, dataset: Option String, body: List String}
  type ParseResult = DataStep | ProcStep

  sort ComposableFunction = SASStatement -> ParseResult

  % Operations
  op compose : [a,b,c] (b -> c) -> (a -> b) -> (a -> c)
  op data_func : SASStatement -> DataStep
  op proc_func : SASStatement -> ProcStep
  op parse_chain : SASStatement -> ParseResult

  % Function composition operation
  axiom compose_def is [a,b,c]
    fa(f: b -> c, g: a -> b, x: a)
      compose f g x = f(g(x))

  % Associativity of composition
  axiom compose_assoc is [a,b,c,d]
    fa(f: c -> d, g: b -> c, h: a -> b)
      compose f (compose g h) = compose (compose f g) h

  % Identity element
  op id : [a] a -> a
  axiom id_def is [a] fa(x: a) id x = x

  axiom compose_id_left is [a,b]
    fa(f: a -> b) compose id f = f

  axiom compose_id_right is [a,b]
    fa(f: a -> b) compose f id = f

end-spec
```

2. SAS-FP2 Parser

perl

```perl
#!/usr/bin/perl
use strict;
use warnings;
use Parse::RecDescent;
use Data::Dumper;

# Enable parse tree - this creates [@item] for each rule
$::RD_AUTOACTION = q { [@item] };

my $grammar = q{
    program: statement(s) eof

    statement: (data_step | proc_step)(s?)

    data_step: 'data' identifier options(?) ';'
               (data_body(s) ('run' ';')(?) | ('run' ';')(? | ))

    proc_step: 'proc' proc_name identifier(?) options(?) ';'
               proc_body(s?) ('run' ';')(?)

    # Fixed: Only specific data step statements - NO catch-all rules
    data_body: assignment | conditional | output_stmt | keep_stmt | drop_stmt | label_stmt

    assignment: identifier '=' expression ';'

    output_stmt: 'output' dataset_list(?) condition(?) ';'

    conditional: 'if' condition 'then' statement_block
                 ('else' statement_block)(?)

    # Common data step statements
    keep_stmt: 'keep' identifier(s) ';'
    drop_stmt: 'drop' identifier(s) ';'
    label_stmt: 'label' identifier '=' /[^;]+/ ';'

    proc_body: proc_option | proc_statement_line

    # Lexical elements
    identifier: /[a-zA-Z_][a-zA-Z0-9_]*/
    proc_name: /[a-zA-Z][a-zA-Z0-9]*/
    expression: /[^;]+/
    condition: /[^;]+/
    dataset_list: identifier(s /\s+/)
    options: /\([^)]*\)/
```

```
    statement_block: /[^;]+;/
    proc_option: /\(([^)]+\);?/ | /[a-zA-Z_][a-zA-Z0-9_]*\s*=\s*[^;]+;/
    proc_statement_line: /[^;]+;/
    eof: /^\s*\Z/
};

my $parser = Parse::RecDescent->new($grammar);
if (!defined $parser) {
    die "Parser creation failed!";
}

sub get_parser {
    return $parser;
}
```

3. Perl Function Composition Engine with Chained Subroutines

```perl
package FunctionComposition;
use strict;
use warnings;
use Exporter 'import';

our @EXPORT_OK = qw(compose data_func proc_func parse_chain);

# Core composition function - implements (f ∘ g)(x) = f(g(x))
sub compose {
    my ($f, $g) = @_;
    return sub {
        my @args = @_;
        return $f->($g->(@args));
    };
}

# Data step processing function
sub data_func {
    my ($input) = @_;
    my $parser = main::get_parser();

    # Extract data step information
    if ($input =~ /data\s+(\w+)/) {
        return {
            type => 'data',
            name => $1,
            parsed => $parser->program($input)
        };
    }
    return undef;
}

# Proc step processing function
sub proc_func {
    my ($input) = @_;
    my $parser = main::get_parser();

    # Extract proc information
    if ($input =~ /proc\s+(\w+)(?:\s+(\w+))?/) {
        return {
            type => 'proc',
            proc_name => $1,
            dataset => $2,
            parsed => $parser->program($input)
```

```perl
        };
    }
    return undef;
}

# Main parsing chain: proc ∘ data
sub parse_chain {
    my ($input) = @_;

    # Create composition chain
    my $proc_to_data = compose(\&proc_func, \&data_func);

    return $proc_to_data->($input);
}

# Alternative direct composition for complex statements
sub compose_all {
    my ($input) = @_;

    # For statements like "data crime1; proc means; proc print crime2;"
    my @results;

    # Split into individual statements
    my @statements = split /;/, $input;

    for my $stmt (@statements) {
        $stmt =~ s/^\s+|\s+$//g; # trim whitespace
        next if $stmt eq '';

        if ($stmt =~ /^data/) {
            push @results, data_func($stmt . ';');
        } elsif ($stmt =~ /^proc/) {
            push @results, proc_func($stmt . ';');
        }
    }

    return \@results;
}

1;
```

4. SML Code for the Function Composition Engine

sml

```
(* SML Function Composition Engine for SAS-FP2 *)

datatype sas_statement =
    DataStep of string * string list
  | ProcStep of string * string option * string list

datatype parse_result =
    ParseSuccess of sas_statement
  | ParseFailure of string

(* Function composition operator *)
infix o
fun (f o g) x = f (g x)

(* Basic parsing functions *)
fun data_func input =
    case String.tokens (fn c => c = #" ") input of
        ["data", name] => ParseSuccess (DataStep (name, []))
      | _ => ParseFailure "Invalid data step"

fun proc_func input =
    case String.tokens (fn c => c = #" ") input of
        ["proc", proc_name] => ParseSuccess (ProcStep (proc_name, NONE, []))
      | ["proc", proc_name, dataset] => ParseSuccess (ProcStep (proc_name, SOME dataset, []))
      | _ => ParseFailure "Invalid proc step"

(* Composition chain: proc ∘ data *)
val parse_chain = proc_func o data_func

(* Helper function for identity *)
fun id x = x

(* Composition properties *)
fun compose_assoc f g h = (f o (g o h)) = ((f o g) o h)
fun compose_id_left f = (id o f) = f
fun compose_id_right f = (f o id) = f

(* Example usage *)
fun test_composition () =
    let
        val input1 = "data crime1"
        val input2 = "proc means crime1"
        val input3 = "proc print crime2"
```

```
    val result1 = data_func input1
    val result2 = proc_func input2
    val result3 = proc_func input3
    val result4 = parse_chain input1
  in
    (result1, result2, result3, result4)
  end
```

5. Specware Axioms and Theorems in Isabelle/HOL

isabelle

```
theory SAS_FP2_Composition
  imports Main
begin

(* Type definitions *)
datatype sas_statement = DataStep string "string list"
                       | ProcStep string "string option" "string list"

datatype parse_result = Success sas_statement | Failure string

(* Function composition *)
definition compose :: "('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c" (infixl "∘" 55) where
"compose f g ≡ λx. f (g x)"

(* Parser functions *)
consts
  data_func :: "string ⇒ parse_result"
  proc_func :: "string ⇒ parse_result"

(* Composition axioms *)
lemma compose_assoc: "(f ∘ g) ∘ h = f ∘ (g ∘ h)"
  by (simp add: compose_def fun_eq_iff)

lemma compose_id_left: "id ∘ f = f"
  by (simp add: compose_def)

lemma compose_id_right: "f ∘ id = f"
  by (simp add: compose_def)

(* Parsing chain theorem *)
definition parse_chain :: "string ⇒ parse_result" where
"parse_chain ≡ proc_func ∘ data_func"

(* Correctness theorem *)
theorem parse_chain_correctness:
  assumes "valid_sas_input input"
  shows "∃result. parse_chain input = Success result"
  sorry (* Proof would require concrete parser implementation *)

(* Composition properties for SAS parsing *)
theorem sas_parse_associativity:
  "proc_func ∘ data_func = proc_func ∘ data_func"
  by simp

(* Determinism theorem *)
```

```
(* Determinism theorem *)
theorem parse_determinism:
  assumes "parse_chain input = Success result1"
  assumes "parse_chain input = Success result2"
  shows "result1 = result2"
  by simp

(* Composability theorem *)
theorem function_composability:
  fixes f :: "'b ⇒ 'c" and g :: "'a ⇒ 'b" and h :: "'d ⇒ 'a"
  shows "f ∘ (g ∘ h) = (f ∘ g) ∘ h"
  by (rule compose_assoc)

end
```

Summary

This comprehensive specification integrates:

1. **Formal Specware types and operations** for function composition
2. **Modified SAS-FP2 parser** with unified proc handling
3. **Perl composition engine** implementing chained subroutines
4. **SML functional composition** using the `o` operator
5. **Isabelle/HOL formal proofs** of composition properties

The key innovation is treating SAS statement parsing as function composition: `print ∘ (proc ∘ data)`, where each function transforms and refines the parse tree, following both functional programming principles and formal specification methodology.