Author: Gabriel Rovesti

Introduction

This document provides a rigorous mathematical foundation for understanding programming language semantics, with a focus on functional languages. We develop the formal tools necessary to precisely define, analyze, and reason about programming languages - particularly their syntax, operational semantics, and type systems.

Part 1: Mathematical Foundations of Syntax

1.1 Abstract Syntax

Abstract syntax trees (ASTs) provide a mathematical representation of program structure that abstracts away from concrete syntax details.

Definition 1.1.1: An abstract syntax tree is a labeled tree where:

- Internal nodes represent operations or language constructs
- Leaf nodes represent atomic values or variables
- The structure represents the hierarchical organization of a program

Definition 1.1.2: The abstract syntax for our functional language is given by the following grammar:

```
e ::= x                          (variable)
    | λx.e                       (abstraction)
    | e₁ e₂                      (application)
    | n                          (integer literal)
    | e₁ + e₂                    (addition)
    | true | false               (boolean literals)
    | if e₁ then e₂ else e₃      (conditional)
    | let x = e₁ in e₂           (let binding)
    | let rec f = λx.e₁ in e₂    (recursive let binding)
    | (e₁, ..., eₙ)              (tuple)
    | πᵢ e                       (projection)
```

Note that the abstract syntax is distinct from concrete syntax, which includes parsing concerns and notation like parentheses.

1.2 Structural Induction

Structural induction is a proof technique for demonstrating properties of inductively defined structures like ASTs.

Principle of Structural Induction: To prove that a property P holds for all terms in an inductively defined set S:

1. Base case: Prove P holds for all basic elements in S
2. Inductive step: Assume P holds for subterms, then prove it holds for terms constructed from them

1.3 Variable Binding and Scope

Definition 1.3.1: In an expression λx.e, the variable x is bound in e. A variable that is not bound is free.

Definition 1.3.2: The set of free variables in an expression e, denoted FV(e), is defined inductively:

- $FV(x) = \{x\}$
- $FV(\lambda x.e) = FV(e) \setminus \{x\}$
- $FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$
- $FV(n) = \emptyset$
- $FV(e_1 + e_2) = FV(e_1) \cup FV(e_2)$
- $FV(\text{true}) = FV(\text{false}) = \emptyset$
- $FV(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = FV(e_1) \cup FV(e_2) \cup FV(e_3)$
- $FV(\text{let } x = e_1 \text{ in } e_2) = FV(e_1) \cup (FV(e_2) \setminus \{x\})$
- $FV(\text{let rec } f = \lambda x.e_1 \text{ in } e_2) = (FV(e_1) \setminus \{x, f\}) \cup (FV(e_2) \setminus \{f\})$
- $FV((e_1, ..., e_n)) = FV(e_1) \cup ... \cup FV(e_n)$
- $FV(\pi_i\ e) = FV(e)$

Definition 1.3.3: An expression with no free variables is closed.

1.4 Alpha-Equivalence and Substitution

Definition 1.4.1: Two expressions are alpha-equivalent if one can be obtained from the other by consistent renaming of bound variables.

Definition 1.4.2: Substitution of term s for variable x in term t, denoted t[s/x], is defined inductively with the following constraints:

1. $x[s/x] = s$
2. $y[s/x] = y$ if $y \neq x$
3. $(t_1\ t_2)[s/x] = (t_1[s/x])\ (t_2[s/x])$
4. $(\lambda y.t)[s/x] = \lambda y.(t[s/x])$ if $y \neq x$ and $y \notin FV(s)$
5. $(\lambda y.t)[s/x] = \lambda y.t$ if $y = x$

6. $(\lambda y.t)[s/x] = \lambda z.((t[z/y])[s/x])$ if $y \neq x$, $y \in FV(s)$, and z is fresh
7. $(n)[s/x] = n$
8. $(t_1 + t_2)[s/x] = (t_1[s/x]) + (t_2[s/x])$
9. $(\text{true})[s/x] = \text{true}$, $(\text{false})[s/x] = \text{false}$
10. $(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)[s/x] = \text{if } t_1[s/x] \text{ then } t_2[s/x] \text{ else } t_3[s/x]$
11. $(\text{let } y = t_1 \text{ in } t_2)[s/x] = \text{let } y = t_1[s/x] \text{ in } t_2[s/x]$ if $y \neq x$ and $y \notin FV(s)$
12. $(\text{let } y = t_1 \text{ in } t_2)[s/x] = \text{let } y = t_1[s/x] \text{ in } t_2$ if $y = x$
13. $(\text{let } y = t_1 \text{ in } t_2)[s/x] = \text{let } z = t_1[s/x] \text{ in } ((t_2[z/y])[s/x])$ if $y \neq x$, $y \in FV(s)$, and z is fresh

The use of a fresh variable z is justified by the need for capture-avoiding substitution (Barendregt's convention), which ensures that free variables in s don't become accidentally bound during substitution.

Theorem 1.4.3: If e_1 and e_2 are alpha-equivalent, then $e_1[s/x]$ and $e_2[s/x]$ are alpha-equivalent, provided that s is capture-avoiding.

Part 2: Operational Semantics

2.1 Call-by-Value Small-Step Operational Semantics

Small-step semantics defines program execution as a sequence of atomic steps. We adopt a call-by-value (CBV) evaluation strategy.

Definition 2.1.1: A small-step semantics is defined by a relation $e \to e'$ indicating that expression e reduces in one step to e'.

For our language, the reduction rules are:

1. **Beta-reduction** (CBV): $(\lambda x.e_1)\ v \to e_1[v/x]$ where v is a value
2. **Arithmetic**: $n_1 + n_2 \to n_3$ where n_3 is the sum of n_1 and n_2
3. **Conditional**:
   - $\text{if true then } e_2 \text{ else } e_3 \to e_2$
   - $\text{if false then } e_2 \text{ else } e_3 \to e_3$
4. **Let binding**: $\text{let } x = v \text{ in } e \to e[v/x]$ where v is a value
5. **Recursive let**: $\text{let rec } f = \lambda x.e_1 \text{ in } e_2 \to e_2[\langle \lambda x.e_1; f; \Delta \rangle / f]$ where $\langle \lambda x.e_1; f; \Delta \rangle$ is a value (recursive closure)
6. **Tuple projection**: $\pi_i(v_1, ..., v_n) \to v_i$ where each v_k is a value

Additionally, we must define evaluation contexts to specify the order of evaluation:

```
E ::= □                 (hole)
    | E e               (application left)
    | v E               (application right)
    | E + e             (addition left)
```

```
    | v + E             (addition right)
    | if E then e else e (conditional guard)
    | let x = E in e     (let binding)
    | (v₁, ..., vᵢ₋₁, E, eᵢ₋₁, ..., eₙ) (tuple component)
    | πᵢ E               (projection)
```

With the context rule:

```
e → e'
-------
E[e] → E[e']
```

Definition 2.1.2: The reflexive, transitive closure of →, denoted →*, represents multi-step reduction.

Definition 2.1.3: An expression e is in normal form if there is no e' such that e → e'.

Theorem 2.1.4 (Progress): For a closed, well-typed expression e, either e is a value or there exists e' such that e → e'.

Theorem 2.1.5 (Preservation): If e is well-typed with type T and e → e', then e' is also well-typed with type T.

2.2 Basic Theory of Abstract Reduction Systems

An abstract reduction system generalizes the concept of computation steps.

Definition 2.2.1: An abstract reduction system (ARS) is a pair (A, →) where A is a set and → ⊆ A × A is a binary relation on A.

Definition 2.2.2: A reduction sequence in an ARS is a sequence $a_0 \to a_1 \to a_2 \to \ldots$

Definition 2.2.3: An element a ∈ A is:

- Reducible if there exists b such that a → b
- In normal form if it is not reducible
- Weakly normalizing if there exists a reduction sequence from a to a normal form
- Strongly normalizing if every reduction sequence from a is finite

Definition 2.2.4: An ARS is:

- Confluent if for all a, b, c with a → *b and a* → c, there exists d such that b → *d and c* → d
- Church-Rosser if for all a, b with a ↔ *b, there exists c such that a* → c and b →* c
- Terminating if there are no infinite reduction sequences

Theorem 2.2.5: An ARS is confluent if and only if it has the Church-Rosser property.

Theorem 2.2.6 (Diamond Property): If an ARS has the property that for all a, b, c with a → b and a → c, there exists d such that b → d and c → d, then it is confluent.

2.3 Environments and Closures

To properly model lexical scoping, we use environments and closures.

Definition 2.3.1: An evaluation environment Δ is a mapping from variables to values:

```
Δ ::= ∅ | Δ,(x ↛ v)
```

Definition 2.3.2: A closure ⟨λx.e; Δ⟩ captures a lambda abstraction together with the environment at the time of definition.

Definition 2.3.3: A recursive closure ⟨λx.e; f; Δ⟩ additionally captures the name of the recursive function.

2.4 Values

Values are the results of evaluation and are defined as:

```
v ::= L                  (literal)
    | ⟨λx.e; Δ⟩          (closure)
    | ⟨λx.e; f; Δ⟩       (recursive closure)
    | (v₁, ..., vₙ)      (tuple of values)
```

where L includes integers, floats, booleans, strings, characters, and unit.

2.5 Program Equivalence

Definition 2.5.1: Two expressions e_1 and e_2 are operationally equivalent, denoted $e_1 \cong e_2$, if for all contexts C, $C[e_1] \to v$ *if and only if* $C[e_2] \to v$ for some value v.

Definition 2.5.2: A context C is an expression with a "hole" □ that can be filled with an expression.

Theorem 2.5.3: Beta-equivalence: (λx.e) v ≅ e[v/x] where v is a value.

Theorem 2.5.4: Eta-equivalence: λx.(e x) ≅ e if x is not free in e.

Part 3: Type Systems

3.1 Simple Types

Definition 3.1.1: The grammar of simple types:

```
τ ::= c                (type constructor)
    | τ₁ → τ₂          (arrow type)
    | α, β, γ, ..      (type variables)
    | τ₁ * ... * τₙ    (tuple type)
```

where c represents type names like int, float, bool, etc.

Definition 3.1.2: A typing context Γ is a finite mapping from variables to types:

```
Γ ::= ∅ | Γ,(x : τ)
```

Definition 3.1.3: The typing judgment Γ ⊢ e : τ indicates that expression e has type τ in context Γ.

3.1.4 Typing Rules

1. Variable:

```
x ∈ dom(Γ)
Γ(x) = τ
---------
Γ ⊢ x : τ
```

2. Abstraction:

```
Γ, x:τ₁ ⊢ e : τ₂
-----------------
Γ ⊢ λx.e : τ₁ → τ₂
```

3. Application:

```
Γ ⊢ e₁ : τ₁ → τ₂    Γ ⊢ e₂ : τ₁
-------------------------------
Γ ⊢ e₁ e₂ : τ₂
```

4. Integer:

```
-----------
Γ ⊢ n : int
```

5. Addition:

```
Γ ⊢ e₁ : int    Γ ⊢ e₂ : int
---------------------------
Γ ⊢ e₁ + e₂ : int
```

6. Boolean:

```
---------------
Γ ⊢ true : bool
```

```
----------------
Γ ⊢ false : bool
```

7. Conditional:

```
Γ ⊢ e₁ : bool    Γ ⊢ e₂ : τ    Γ ⊢ e₃ : τ
-----------------------------------------
Γ ⊢ if e₁ then e₂ else e₃ : τ
```

8. Let binding:

```
Γ ⊢ e₁ : τ₁    Γ, x:τ₁ ⊢ e₂ : τ₂
-------------------------------
Γ ⊢ let x = e₁ in e₂ : τ₂
```

9. Recursive let:

```
Γ, f:τ₁→τ₂ ⊢ λx.e₁ : τ₁→τ₂    Γ, f:τ₁→τ₂ ⊢ e₂ : τ₃
-------------------------------------------------
Γ ⊢ let rec f = λx.e₁ in e₂ : τ₃
```

10. Tuple:

```
Γ ⊢ e₁ : τ₁  ...  Γ ⊢ eₙ : τₙ
----------------------------
Γ ⊢ (e₁, ..., eₙ) : τ₁ * ... * τₙ
```

11. Projection:

```
Γ ⊢ e : τ₁ * ... * τₙ    1 ≤ i ≤ n
------------------------------
Γ ⊢ πᵢ e : τᵢ
```

Theorem 3.1.5 (Type Safety): A well-typed program never gets "stuck" (i.e., reaches an invalid state during execution). This follows from progress and preservation.

3.2 Polymorphic Types and Let-Polymorphism

Definition 3.2.1: A type scheme σ is a type with universally quantified type variables:

```
σ ::= ∀α.τ       (type scheme)
```

The typing environment Γ now maps identifiers to type schemes:

```
Γ ::= ∅ | Γ,(x : σ)
```

3.2.2 Auxiliary Functions

Several auxiliary functions are needed for polymorphic type systems:

1. Free type variables (ftv):

```
ftv : (τ ∪ σ ∪ Γ) → P(α)
ftv(c) = ∅
ftv(α) = {α}
ftv(τ₁ → τ₂) = ftv(τ₁) ∪ ftv(τ₂)
ftv(τ₁ * ... * τₙ) = ftv(τ₁) ∪ ... ∪ ftv(τₙ)
ftv(∀α.τ) = ftv(τ) \ {α}
ftv(∅) = ∅
ftv(Γ,(x : σ)) = ftv(σ) ∪ ftv(Γ)
```

2. Generalization (gen):

```
gen : Γ × τ → σ
genΓ(τ) = ∀α.τ  where α = ftv(τ) \ ftv(Γ)
```

3. Instantiation (inst):

```
inst : σ → τ
inst(∀α.τ) = rea(τ)
```

4. Type variable refreshing (re):

```
re : P(α) × τ → τ
rea(c) = c
rea(α) = α  if α ∉ α
rea(α) = β  if α ∈ α and β is fresh
rea(τ₁ → τ₂) = rea(τ₁) → rea(τ₂)
rea(τ₁ * ... * τₙ) = rea(τ₁) * ... * rea(τₙ)
```

3.2.3 Polymorphic Typing Rules

1. Variable:

```
x ∈ dom(Γ)    Γ(x) = σ    τ = inst(σ)
-------------------------------------
Γ ⊢ x : τ
```

2. Let binding (with polymorphism):

```
Γ ⊢ e₁ : τ₁    σ₁ = genΓ(τ₁)    Γ,(x : σ₁) ⊢ e₂ : τ₂
---------------------------------------------------
Γ ⊢ let x = e₁ in e₂ : τ₂
```

3.3 Type Inference (Hindley-Milner)

Definition 3.3.1: A substitution θ is a finite mapping from type variables to types:

```
θ ::= ∅ | θ,[α ↦ τ]
```

Definition 3.3.2: Substitution application θ(τ) is defined inductively:

```
θ(c) = c
θ(α) = τ  if [α ↦ τ] ∈ θ
θ(α) = α  if α ∉ dom(θ)
θ(τ₁ → τ₂) = θ(τ₁) → θ(τ₂)
θ(τ₁ * ... * τₙ) = θ(τ₁) * ... * θ(τₙ)
```

Definition 3.3.3: Substitution application to schemes and environments:

```
θ(∀α.τ) = ∀α.θ'(τ)  where θ' = θ \ {α ↦ τ | α ∈ α}
θ(∅) = ∅
θ(Γ,(x : σ)) = θ(Γ),(x : θ(σ))
```

Definition 3.3.4: Composition of substitutions $\theta_2 \circ \theta_1$:

```
(θ₂ ∘ θ₁)(τ) = θ₂(θ₁(τ))
```

Definition 3.3.5: The most general unifier (mgu) $U(\tau_1; \tau_2)$ computes a substitution θ such that $\theta(\tau_1) \equiv \theta(\tau_2)$:

```
U(c₁; c₂) = ∅  if c₁ ≡ c₂
U(α; τ) = [α ↦ τ]  if α ∉ ftv(τ)
U(τ; α) = [α ↦ τ]  if α ∉ ftv(τ)
U(τ₁ → τ₂; τ₃ → τ₄) = θ₂ ∘ θ₁  where θ₁ = U(τ₁; τ₃), θ₂ = U(θ₁(τ₂); θ₁(τ₄))
U(τ₁ * ... * τₙ; τ'₁ * ... * τ'ₙ) = θₙ ∘ ... ∘ θ₁  where θᵢ = U(θᵢ₋₁ ∘ ...
∘ θ₁(τᵢ); θᵢ₋₁ ∘ ... ∘ θ₁(τ'ᵢ))
```

3.3.6 Hindley-Milner Type Inference Algorithm

The type inference judgment $\Gamma \vdash e : \tau \triangleright \theta$ produces both a type τ and a substitution θ.

1. Literal:

   ```
   Γ ⊢ n : int ▷ ∅
   ```

2. Variable:

   ```
   x ∈ dom(Γ)    Γ(x) = σ    τ = inst(σ)
   ------------------------------------
   Γ ⊢ x : τ ▷ ∅
   ```

3. Abstraction:

   ```
   Γ,(x : ∀∅.α) ⊢ e : τ₂ ▷ θ₁    τ₁ = θ₁(α)    (α fresh)
   ------------------------------------------------
   Γ ⊢ λx.e : τ₁ → τ₂ ▷ θ₁
   ```

4. Application:

```
Γ ⊢ e₁ : τ₁ ▷ θ₁    θ₁(Γ) ⊢ e₂ : τ₂ ▷ θ₂
U(θ₂(τ₁); τ₂ → α) = θ₃    (α fresh)
τ = θ₃(α)    θ = θ₃ ∘ θ₂ ∘ θ₁
-------------------------------------
Γ ⊢ e₁ e₂ : τ ▷ θ
```

5. Let binding:

```
Γ ⊢ e₁ : τ₁ ▷ θ₁    σ₁ = genθ₁(Γ)(τ₁)
θ₁(Γ),(x : σ₁) ⊢ e₂ : τ₂ ▷ θ₂    θ = θ₂ ∘ θ₁
-----------------------------------------------
Γ ⊢ let x = e₁ in e₂ : τ₂ ▷ θ
```

6. Recursive let:

```
Γ,(f : ∀∅.α) ⊢ λx.e₁ : τ₁ ▷ θ₁    Γ₁ = θ₁(Γ)    σ₁ = genΓ₁(τ₁)    (α
fresh)
Γ₁,(f : σ₁) ⊢ e₂ : τ₂ ▷ θ₂    U(α; θ₁(τ₁)) = θ₃    θ = θ₃ ∘ θ₂ ∘ θ₁
-------------------------------------------------------------------------
---
Γ ⊢ let rec f = λx.e₁ in e₂ : τ₂ ▷ θ
```

Example: Step-by-step derivation for the identity function:

1. Start with: `id = λx.x`
2. For `λx.x`:
   - Introduce fresh type variable α for x: Γ = {x : α}
   - Type the body: Γ ⊢ x : α ▷ ∅
   - By abstraction rule: ∅ ⊢ λx.x : α → α ▷ ∅
3. Final type: `id : α → α` (polymorphic)

Part 4: Practical Examples in Standard ML

4.1 Example: Higher-Order Functions

Higher-order functions take functions as parameters or return them as results.

```
(* Map function over lists *)
fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs;

(* Filter function *)
```

```
fun filter p [] = []
  | filter p (x::xs) = if p x then x :: filter p xs else filter p xs;

(* Fold function (right-associative) *)
fun foldr f z [] = z
  | foldr f z (x::xs) = f x (foldr f z xs);

(* Fold function (left-associative) *)
fun foldl f z [] = z
  | foldl f z (x::xs) = foldl f (f z x) xs;

(* Examples *)
val squares = map (fn x => x * x) [1, 2, 3, 4, 5];
val evens = filter (fn x => x mod 2 = 0) [1, 2, 3, 4, 5];
val sum = foldr (op +) 0 [1, 2, 3, 4, 5];
val factorial = foldl (op *) 1 [1, 2, 3, 4, 5];
```

4.2 Example: Tree Operations

```
(* Binary tree type *)
datatype 'a tree = Empty | Node of 'a * 'a tree * 'a tree;

(* Map over tree *)
fun mapTree f Empty = Empty
  | mapTree f (Node(x, left, right)) =
      Node(f x, mapTree f left, mapTree f right);

(* Fold over tree *)
fun foldTree f z Empty = z
  | foldTree f z (Node(x, left, right)) =
      f x (foldTree f z left) (foldTree f z right);

(* Examples *)
val myTree = Node(5,
                  Node(3, Node(1, Empty, Empty), Empty),
                  Node(7, Empty, Node(9, Empty, Empty)));

val doubledTree = mapTree (fn x => x * 2) myTree;
val sum = foldTree (fn x => fn left => fn right => x + left + right) 0
myTree;
```

Part 5: Advanced Topics

5.1 Linear Types and Substructural Type Systems

Linear types control resource usage by ensuring that variables are used exactly once.

Definition 5.1.1: In a linear type system, each variable must be used exactly once.

Definition 5.1.2: In a substructural type system, the structural rules may be restricted:

- Weakening: Introduction of unused variables (forbidden in affine systems)
- Contraction: Using variables multiple times (forbidden in linear systems)
- Exchange: Changing the order of variables (forbidden in ordered systems)

Linear Typing Rules:

1. Variable (Linear):

```
-----------
x:τ ⊢ x : τ
```

2. Abstraction (Linear):

```
Γ, x:τ₁ ⊢ e : τ₂
-----------------
Γ ⊢ λx.e : τ₁ ⊸ τ₂
```

   where $\tau_1 \multimap \tau_2$ is a linear function type.
3. Application (Linear):

```
Γ₁ ⊢ e₁ : τ₁ ⊸ τ₂    Γ₂ ⊢ e₂ : τ₁
--------------------------------
Γ₁, Γ₂ ⊢ e₁ e₂ : τ₂
```

Usage Counting Property: In a well-typed program under a linear type system, for any execution path, each resource (variable) is used exactly once - neither duplicated nor discarded. This can be proved by tracking a usage count for each variable and showing that evaluation preserves this count.

5.2 Lazy Evaluation and Call-by-Name

Definition 5.2.1: Different evaluation strategies:

- Call-by-value (eager): Arguments are evaluated before function application
- Call-by-name: Arguments are passed unevaluated and evaluated when used
- Lazy evaluation: Call-by-name with sharing (memoization) of evaluated arguments

The key modification for call-by-name semantics is in the β-reduction rule:

```
(λx.e₁) e₂ → e₁[e₂/x]
```

Notice that e_2 is substituted directly without being evaluated first.

For true lazy evaluation, we need thunk memoization to ensure that e_2 is evaluated at most once, regardless of how many times x is used in e_1. This is typically implemented using a mechanism like thunks that store the result of the first evaluation.

5.3 Mechanized Semantics with Coq

Mechanized semantics uses proof assistants like Coq to formalize language semantics and verify properties.

Here's a Coq formalization of lambda calculus, incorporating the necessary shift/de Bruijn indexing for correct substitution:

```
Inductive term : Type :=
  | var : nat -> term
  | app : term -> term -> term
  | abs : term -> term.

(* Lifting operation to handle de Bruijn indices correctly *)
Fixpoint lift (d:nat) (c:nat) (t:term) : term :=
  match t with
  | var n => if le_gt_dec c n then var (n+d) else var n
  | app t1 t2 => app (lift d c t1) (lift d c t2)
  | abs t' => abs (lift d (S c) t')
  end.

(* Substitution with proper handling of indices *)
Fixpoint subst (s:term) (j:nat) (t:term) : term :=
  match t with
  | var n => match nat_compare n j with
             | Eq => lift j 0 s
             | Gt => var (pred n)
             | Lt => var n
             end
  | app t1 t2 => app (subst s j t1) (subst s j t2)
  | abs t' => abs (subst s (S j) t')
  end.

Inductive step : term -> term -> Prop :=
  | step_beta : forall t1 t2,
      step (app (abs t1) t2) (subst t2 0 t1)
  | step_app1 : forall t1 t1' t2,
      step t1 t1' ->
      step (app t1 t2) (app t1' t2)
  | step_app2 : forall v t2 t2',
      value v ->
      step t2 t2' ->
      step (app v t2) (app v t2')
```

```
with value : term -> Prop :=
  | value_abs : forall t, value (abs t).
```

Part 6: Standard ML Implementation Examples

6.1 Implementing an Interpreter

```
(* Expression type for a simple language *)
datatype expr =
    Var of string
  | Int of int
  | Bool of bool
  | Add of expr * expr
  | If of expr * expr * expr
  | Fun of string * expr
  | App of expr * expr;

(* Value type *)
datatype value =
    IntVal of int
  | BoolVal of bool
  | FunVal of string * expr * env
and env = (string * value) list;

(* Environment lookup *)
fun lookup [] x = raise Fail ("Unbound variable: " ^ x)
  | lookup ((y, v)::rest) x = if x = y then v else lookup rest x;

(* Evaluation function *)
fun eval _ (Int n) = IntVal n
  | eval _ (Bool b) = BoolVal b
  | eval env (Var x) = lookup env x
  | eval env (Add(e1, e2)) =
      (case (eval env e1, eval env e2) of
           (IntVal n1, IntVal n2) => IntVal(n1 + n2)
         | _ => raise Fail "Type error in addition")
  | eval env (If(e1, e2, e3)) =
      (case eval env e1 of
           BoolVal true => eval env e2
         | BoolVal false => eval env e3
         | _ => raise Fail "Type error in conditional")
  | eval env (Fun(x, body)) = FunVal(x, body, env)
  | eval env (App(e1, e2)) =
      (case eval env e1 of
           FunVal(x, body, env') =>
             let val v = eval env e2
             in eval ((x, v)::env') body
```

```
            end
        | _ => raise Fail "Type error in application");
```

6.2 Type Checker Implementation

```
(* Type for our simple language *)
datatype typ =
    TInt
  | TBool
  | TFun of typ * typ;

(* Type environment *)
type tenv = (string * typ) list;

(* Type lookup *)
fun tlookup [] x = NONE
  | tlookup ((y, t)::rest) x = if x = y then SOME t else tlookup rest x;

(* Type checking function *)
fun typeOf env (Int _) = TInt
  | typeOf env (Bool _) = TBool
  | typeOf env (Var x) =
      (case tlookup env x of
           SOME t => t
         | NONE => raise Fail ("Unbound variable: " ^ x))
  | typeOf env (Add(e1, e2)) =
      (case (typeOf env e1, typeOf env e2) of
           (TInt, TInt) => TInt
         | _ => raise Fail "Type error in addition")
  | typeOf env (If(e1, e2, e3)) =
      (case typeOf env e1 of
           TBool =>
             let val t2 = typeOf env e2
                 val t3 = typeOf env e3
             in if t2 = t3 then t2
                else raise Fail "Branches have different types"
             end
         | _ => raise Fail "Condition must be boolean")
  | typeOf env (Fun(x, body)) =
      let val argType = TInt (* Assume int for simplicity *)
          val bodyType = typeOf ((x, argType)::env) body
      in TFun(argType, bodyType)
      end
  | typeOf env (App(e1, e2)) =
      (case typeOf env e1 of
           TFun(t1, t2) =>
             if typeOf env e2 = t1 then t2
```

```
            else raise Fail "Argument type mismatch"
        | _ => raise Fail "Function expected");
```

Part 7: Functional Programming with F#

7.1 Higher-Order Functions

```
// Map implementation
let rec map f list =
    match list with
    | [] -> []
    | x::xs -> f x :: map f xs

// Filter implementation
let rec filter predicate list =
    match list with
    | [] -> []
    | x::xs when predicate x -> x :: filter predicate xs
    | _::xs -> filter predicate xs

// Fold-left implementation
let rec foldl folder state list =
    match list with
    | [] -> state
    | x::xs -> foldl folder (folder state x) xs

// Fold-right implementation
let rec foldr folder state list =
    match list with
    | [] -> state
    | x::xs -> folder x (foldr folder state xs)

// Examples
let numbers = [1..10]
let squares = map (fun x -> x * x) numbers
let evens = filter (fun x -> x % 2 = 0) numbers
let sum = foldl (+) 0 numbers
let product = foldr (*) 1 numbers
```

7.2 Tree Operations

```
// Binary tree type
type Tree<'a> =
    | Leaf of 'a option
    | Node of Tree<'a> * Tree<'a>

// Map over tree
```

```
let rec mapTree f tree =
    match tree with
    | Leaf None -> Leaf None
    | Leaf (Some x) -> Leaf (Some (f x))
    | Node (left, right) -> Node (mapTree f left, mapTree f right)

// Filter over tree
let rec filterTree predicate tree =
    match tree with
    | Leaf None -> Leaf None
    | Leaf (Some x) -> if predicate x then Leaf (Some x) else Leaf None
    | Node (left, right) -> Node (filterTree predicate left, filterTree
predicate right)

// Fold over tree
let rec foldTree folder state tree =
    match tree with
    | Leaf None -> state
    | Leaf (Some x) -> folder state x
    | Node (left, right) ->
        let leftResult = foldTree folder state left
        foldTree folder leftResult right

// Example
let myTree = Node (Node (Leaf (Some 1), Leaf (Some 2)), Leaf (Some 3))
let doubledTree = mapTree (fun x -> x * 2) myTree
let filteredTree = filterTree (fun x -> x > 1) myTree
let sum = foldTree (fun acc x -> acc + x) 0 myTree
```

Conclusion

This document has provided a comprehensive, formal treatment of programming language semantics, focusing on syntax, operational semantics, and type systems. The mathematical foundations established here allow for rigorous reasoning about program behavior, correctness, and safety.

We've covered the essential theory of functional programming languages through a rigorous mathematical lens, illustrated with practical examples in Standard ML and F#. This knowledge forms the basis for understanding, implementing, and reasoning about programming languages.

References

1. Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002
2. Glynn Winskel, *The Formal Semantics of Programming Languages*, MIT Press, 1993
3. Robert Harper, *Practical Foundations for Programming Languages*, Cambridge University Press, 2016

4. Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987
5. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, *Semantics Engineering with PLT Redex*, MIT Press, 2009