

## Supplemental Methods

## S1.1 Form Construction Implementation

# Data Structure Design

A **form** is any well-formed expression in the calculus of indications.  
The `Form` class represents boundary expressions with the following structure:

```
@dataclass
class Form:
    form_type: FormType  # VOID, MARK, ENCLOSURE, JUXTAPOS...
    contents: List[Form] = field(default_factory=list)
    is_marked: bool = False
```

**Design Rationale:** - `form_type` enables pattern matching for reduction rules - `contents` stores nested forms (children) - `is_marked` distinguishes mark from void at the base level

# Constructor Functions

Function	Input	Output	Example
<code>make_void()</code>	None	Empty form	$\emptyset$
<code>make_mark()</code>	None	Single mark	$\langle \rangle$
<code>enclose(f)</code>	Form	Enclosed form	$\langle f \rangle$
<code>juxtapose(a, b, ...)</code>	Forms	Combined form	$abc\dots$

## Form Equality

Two forms are **structurally equal** if: 1. Same form\_type 2. Same is\_marked value 3. Contents are pairwise equal (recursive)

Note: **Structural equality** (same form structure) differs from **semantic equality** (reduction to same **canonical form**—the irreducible representation after all reductions).

## S1.2 Reduction Engine Architecture

# Pattern Matching Strategy

The reduction engine uses a priority-based pattern matching approach:

## 1. Calling Pattern Detection:

- ▶ Check if form is marked enclosure
- ▶ Check if single child is also marked enclosure
- ▶ If so, extract inner content

## 2. Crossing Pattern Detection:

- ▶ Check if form has multiple simple marks in juxtaposition
- ▶ Count marks vs non-mark contents
- ▶ If >1 marks, condense

## 3. Void Elimination:

- ▶ Check for void elements in juxtaposition
- ▶ Remove voids (identity element for AND)

# Reduction Trace Format

Each step in the reduction trace records:

```
@dataclass
class ReductionStep:
    before: Form      # Form before this step
    after: Form       # Form after this step
    rule: ReductionRule # CALLING, CROSSING, or VOID_ELIM
    location: str     # Human-readable description
```

## Recursive Application

For compound forms, reduction applies recursively:

1. Reduce all children first (bottom-up)
2. Then check if parent can be reduced
3. Repeat until stable

## S1.3 Boolean Algebra Verification

# Translation Protocol

To verify Boolean correspondence:

1. **Parse Boolean expression** to AST

2. **Translate AST** to boundary form:

- ▶ TRUE → make\_mark()
- ▶ FALSE → make\_void()
- ▶ NOT(a) → enclose(translate(a))
- ▶ AND(a, b) → juxtapose(translate(a), translate(b))
- ▶ OR(a, b) →  
enclose(juxtapose(enclose(translate(a)),  
enclose(translate(b))))

3. **Reduce** both sides

4. **Compare** canonical forms

# Truth Table Verification

For operations with 2 variables, exhaustive verification:

$a$	$b$	$a \wedge b$	Boundary	Reduced
T	T	T	$\langle \rangle \langle \rangle$	$\langle \rangle$
T	F	F	$\langle \rangle \emptyset$	$\emptyset$
F	T	F	$\emptyset \langle \rangle$	$\emptyset$
F	F	F	$\emptyset \emptyset$	$\emptyset$

## S1.4 Theorem Verification Protocol

## Consequence Verification

Each consequence (C1-C9) verified by:

1. **Construct LHS** using form builders
2. **Construct RHS** using form builders
3. **Reduce both** to canonical form
4. **Assert equality** of canonical forms

## Parametric Testing

For consequences with variables:

- Substitute all combinations of mark/void
- Verify equality holds for each substitution
- Report any counterexamples

# Verification Report Structure

```
@dataclass
class VerificationResult:
    name: str
    status: VerificationStatus # PASSED, FAILED, ERROR
    details: str
    duration: float
```

## S1.5 Visualization Pipeline

## Nested Boundary Rendering

Forms visualized as nested rectangles:

- 1. **Void**: Empty space (no rectangle)
- 2. **Mark**: Single rectangle
- 3. **Enclosure**: Rectangle containing child visualization
- 4. **Juxtaposition**: Side-by-side rectangles

# Layout Algorithm

```
function LAYOUT(form, x, y, width, height):
    if form.is_void():
        return EmptyRegion(x, y, width, height)
    if form.is_mark():
        return Rectangle(x, y, width, height)
    if form.is_enclosure():
        child = LAYOUT(form.contents[0], x+pad, y+pad, width)
        return Rectangle(x, y, width, height) + child
    if form.is_juxtaposition():
        # Divide width among children
        child_width = width / len(form.contents)
        return [LAYOUT(c, x + i*child_width, y, child_width)
                for i, c in enumerate(form.contents)]
```

## Export Formats

- ▶ **PNG**: Raster image for documentation
- ▶ **SVG**: Vector graphics for publication
- ▶ **ASCII**: Text representation for terminals
- ▶ **LaTeX/TikZ**: Direct embedding in papers

## S1.6 Random Form Generation

## Generation Parameters

---

Parameter	Type	Default	Description
max_depth	int	4	Maximum nesting level
max_width	int	3	Maximum children per juxtaposition
p_mark	float	0.3	Probability of generating mark
p_void	float	0.2	Probability of generating void
p_enclose	float	0.25	Probability of enclosure
p_juxtapose	float	0.25	Probability of juxtaposition

---

## Generation Algorithm

```
function RANDOM_FORM(depth, rng):
    if depth == 0:
        return CHOICE([make_void(), make_mark()], rng)

    p = rng.random()
    if p < p_void:
        return make_void()
    elif p < p_void + p_mark:
        return make_mark()
    elif p < p_void + p_mark + p_enclose:
        return enclose(RANDOM_FORM(depth - 1, rng))
    else:
        n = rng.randint(2, max_width)
        return juxtapose(*[RANDOM_FORM(depth - 1, rng) for
```

## Reproducibility

Fixed random seed (42) ensures reproducible experiments:

```
rng = random.Random(42)
forms = [random_form(max_depth=4, rng=rng) for _ in range(5)]
```