Complete Specification and Design for Simile

Anthony Hunt

October 25, 2025

Contents

1 Overview

This document serves as a living specification of the underlying term rewriting system used in the compiler for a modelling-focused programming language. It includes an overview of the language Simile, a user-facing reference, and implementation plans.

1.1 High Level Ideas

General Strategy A basic strategy to optimize set and relational expressions is:

1. Normalize the expression as a set comprehensions
2. Simplify and reorganize conjuncts of the set comprehension body

Intuition The TRS for this language primarily involves lowering collection data type expressions into pointwise boolean quantifications. Breaking down each operation into set builder notation enables a few key actions:

- Quantifications over sets ($\{ x \cdot G \mid P \}$) are naturally separated into generators (G) and (non-generating) predicates (P). For sets, at least one membership operator per top-level conjunction in G will serve as a concrete element generator in generated code. Then, top level disjunctions will select one membership operation to act as a generator, relegating all others to the predicate level. For example, if the rewrite system observes an intersection of the form $\{ x \cdot x \in S \land x \in T \}$, the set construction operation must iterate over at least one of S and T. Then, the other will act as a condition to check every iteration (becoming $\{ x \cdot x \in S \mid x \in T \}$).
- By definition of generators in quantification notation, operations in G must be statements of the form $x \in S$, where x is used in the "element" portion of the set construction. Statements like $x \notin T$ or checking a property $p(x)$ must act like conditions since they do not produce any iterable elements.
- Any boolean expression for conditions may be rewritten as a combination of $\neg, \lor$, and $\land$ expressions. Therefore, by converting all set notation down into boolean notation and then generating code based on set constructor booleans, we can accommodate any form of predicate function.

2 Motivation

3 Language

This section denotes the design and behaviour of Simile.

Much of the language design is inherited from Event-B and the mathematical language definition in [1].

3.1 Lexicon

Since Simile makes extensive use of Event-B-like set notation and otherwise non-ASCII characters, this section contains a list of such characters, translations to ASCII symbols where suitable, and other useful lexing information.

3.1.1 Whitespace

Similar to Python, indentation and newline tokens are significant in Simile. In particular, nested code blocks, as in if-statements, for-loops, and procedure definitions, require the use of indentations to signify the beginning and end. Indentation/newlines may also be used within collection-type comprehensions or enumerations, similar to the style seen in JSON. Comments are single-lined and start with a pound symbol (#).

3.1.2 Identifiers

Identifiers in Simile follow the standard regular expression, using only ASCII letters, underscores, and numbers: $[a-zA-Z\_][a-zA-Z\_0-9]$.

3.1.3 Primitives

Integers and floating point numbers follow a similar style to modern programming languages. The dash character ($-$) may be used in either subtraction or as a unary negation. Superscript numbers may be used for constant exponentiation. Booleans (True/False) follow Python style, with capitalized first letters. Strings are delimited with double-quotes (") and may be multilined.

3.1.4 Expressions

ASCII tokens will be provided as an alternative to unicode format where suitable.

The following are symbols used for expressions:

Token	ASCII Token	Name
$\cdot$	`.`	Center Dot (Quantifier Separator)
,		Comma
:		Colon
;		Semicolon
|		Vertical Bar
λ	`lambda`	Lambda
(		Left Parenthesis
)		Right Parenthesis
$\langle$	`<<`	Left Angle Bracket
	`[`	Left Angle Bracket (Alternative)
$\rangle$	`>>`	Right Angle Bracket
	`]`	Right Angle Bracket (Alternative)

Token	ASCII Token	Name
$\{$		Left Brace
$\}$		Right Brace
$[\![$	`[[`	Left Double Bracket
$]\!]$	`]]`	Right Double Bracket
$=$		Equals
$\neq$	`!=`	Not Equals
$\neg$	`not`	Not
	`!`	Not (Alternative)
$\wedge$	`and`	And
$\vee$	`or`	Or
$\Rightarrow$	`=>`	Implies
$\equiv$	`==`	Equivalent
$\not\equiv$	`!==`	Not Equivalent
$\forall$	`forall`	For All
$\exists$	`exists`	There Exists
	`+`	Plus
	`-`	Minus
	`*`	Multiplication
$\times$	`><`	Cartesian Product
	`/`	Division
	`div`	Integer Division
	`mod`	Modulo
	`^`	Exponent
$<$		Less Than
$\leq$	`<=`	Less Than or Equal
$>$		Greater Than
$\geq$	`>=`	Greater Than or Equal
$\in$	`in`	In (Membership)
$\notin$	`not in`	Not In (Membership)
$\cup$	`\/`	Union
$\cap$	`/\`	Intersection
$\setminus$	`\`	Backslash (Set Minus)
$\subset$	`<<:`	Subset
$\subseteq$	`<:`	Subset or Equal
$\supset$	`:>>`	Superset
$\supseteq$	`:>`	Superset or Equal
$\not\subset$		Not Subset
$\not\subseteq$		Not Subset or Equal
$\not\supset$		Not Superset
$\not\supseteq$		Not Superset or Equal
$\bigcup$	`union`	General Union
$\bigcap$	`intersection`	General Intersection
$\mapsto$	`|->`	Maplet
$\oplus$	`<+>`	Relation Overriding
$\circ$	`circ`	Composition

Token	ASCII Token	Name
$+\!\!+$	`++`	Concatenation
$^{-1}$	`~`	Inverse
$\lhd$	`<|`	Domain Restriction
$\not\lhd$	`<<|`	Domain Subtraction
$\rhd$	`|>`	Range Restriction
$\not\rhd$	`|>>`	Range Subtraction
$\leftrightarrow$	`<->`	Relation
$\twoheadleftarrow\!\!\!\!\leftrightarrow$	`<<->`	Total Relation [1]
$\leftrightarrow\!\!\!\!\twoheadrightarrow$	`<->>`	Surjective Relation [2]
$\twoheadleftarrow\!\!\!\!\leftrightarrow\!\!\!\!\twoheadrightarrow$	`<<->>`	Total Surjective Relation [3]
$\nrightarrow$	`+->`	Partial Function
$\rightarrow$	`->`	Total Function
$\rightarrowtail\!\!\!\!\nrightarrow$	`>+>`	Partial Injection
$\rightarrowtail$	`>->`	Total Injection
$\twoheadrightarrow\!\!\!\!\nrightarrow$	`+->>`	Partial Surjection
$\twoheadrightarrow$	`->>`	Total Surjection
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	`>->>`	Bijection
$..$		Upto
$\mathbb{P}$	`powerset`	Powerset
$\mathbb{P}_1$	`powerset1`	Non-Empty Powerset

The following are symbols used for programming constructs:

(Unicode) Token	(ASCII) Token	Name
$:=$	`:=`	Assignment
$:\in$	`::`	Non-deterministic Membership Assignment
$\rightarrow$	`->`	Right Arrow

3.1.5 Reserved Keywords

The following list of symbols and identifiers are reserved for programming constructs:

true	false	if	else	for
while	record	enum	procedure	is
is not	return	exists	break	continue
from	import	skip	with	type
min	max	choice		

[1] Brave Sans Mono unicode: U+E100
[2] Brave Sans Mono unicode: U+E101
[3] Brave Sans Mono unicode: U+E102

3.2 Syntax

We follow the standard EBNF notation for describing the language structure.

Start ::= Statements

Statements ::= {SimpleStmt | CompoundStmt}

SimpleStmt ::= Expr [[‘:’ Expr] (‘≔’ | ‘:∈’) Expr] NEWLINE [WithStmt]
  | ControlFlowStmt NEWLINE
  | ImportStmt NEWLINE

WithStmt ::= INDENT ‘`with`’ Expr NEWLINE {Expr NEWLINE} DEDENT

ControlFlowStmt ::= ‘`return`’ [Expr]
  | ‘`break`’
  | ‘`continue`’
  | ‘`skip`’

ImportStmt ::= ‘`import`’ STRING
  | ‘`from`’ STRING ‘`import`’ ImportList

ImportList ::= ‘`*`’
  | FlatTupleIdentifier
  | ‘(’ FlatTupleIdentifier ‘)’

FlatTupleIdentifier ::= IDENTIFIER {‘,’ IDENTIFIER }

Expr ::= Quantification | Predicate

Quantification ::= ‘λ’ FlatTupleIdentifier ‘·’ Predicate ‘|’ Expr
  | ‘⋃’ QuantificationBody
  | ‘⋂’ QuantificationBody
  | ‘`min`’ QuantificationBody
  | ‘`max`’ QuantificationBody

QuantificationBody ::= IdentList ‘·’ Predicate ‘|’ Expr
  | Expr ‘|’ Predicate

IdentList ::= IdentListItem {‘,’ IdentListItem}

IdentListItem ::= IDENTIFIER
  | IdentListItem ‘↦’ IdentListItem
  | ‘(’ IdentList ‘)’

Predicate ::= UnquantifiedPredicate
  | ‘∀’ IdentList ‘·’ Predicate
  | ‘∃’ IdentList ‘·’ Predicate

UnquantifiedPredicate ::= Implication
  | UnquantifiedPredicate (‘≡’ | ‘≢’) Implication

Implication ::= Disjunction
  | Disjunction {‘⇒’ Disjunction}

Disjunction ::= Conjunction {‘∨’ Conjunction}

Conjunction ::= Negation {‘∧’ Negation}

Negation ::= AtomBool
  | ‘¬’ Negation

AtomBool ::= PairExpr [(‘=’ | ‘≠’ | ‘<’ | ‘>’ | ‘≤’ | ‘≥’ | ‘∈’ | ‘∉’ | ‘⊂’ | ‘⊆’ |
    ‘⊃’ | ‘⊇’ | ‘⊄’ | ‘⊈’ | ‘⊅’ | ‘⊉’) PairExpr]

PairExpr ::= RelSetExpr
  | PairExpr ‘↦’ RelSetExpr

RelSetExpr ::= SetExpr [(‘↔’ | ‘⇸↠’ | ‘↔’ | ‘⇸’ | ‘↠’ | ‘→’ | ‘⤔’ | ‘⤖’ | ‘⤀’
    | ‘⇸’ | ‘↣’) RelSetExpr]

SetExpr ::= IntervalExpr
  | IntervalExpr {‘∪’ IntervalExpr}
  | IntervalExpr {‘×’ IntervalExpr}
  | IntervalExpr {‘⊕’ IntervalExpr}
  | IntervalExpr {‘∘’ IntervalExpr}
  | IntervalExpr {‘∩’ IntervalExpr}
  | IntervalExpr {‘⧺’ IntervalExpr}
  | IntervalExpr (‘⩤’ | ‘◁’) IntervalExpr {‘∩’ IntervalExpr} [RelSubExpr]

RelSubExpr ::= (‘\’ | ‘⩥’ | ‘⋫’) IntervalExpr

IntervalExpr ::= ArithmeticExpr [‘..’ ArithmeticExpr]

ArithmeticExpr ::= Term
  | ArithmeticExpr (‘+’ | ‘-’) Term

Term ::= Factor
  | Term (‘*’ | ‘/’ | ‘div’ | ‘mod’) Factor

Factor ::= [(‘+’ | ‘-’)] Power

Power ::= Primary [‘^’ Factor | [‘−’](‘*INTEGER*’ | ‘*FLOAT*’)]

Primary ::= Atom
  | Atom ‘.’ IDENTIFIER

| Atom '(' Expr {',' Expr} ')'
| Atom '[' Expr ']'

Atom ::= INTEGER | FLOAT | STRING | BOOLEAN | IDENTIFIER
| Set | Sequence | Bag | Tuple
| 'ℙ' '(' Expr ')' | 'ℙ$_1$' '(' Expr ')' | '(' Expr ')'

Set ::= '{' CollectionBody '}'

Bag ::= '⟦' CollectionBody '⟧'

Sequence ::= '⟨' CollectionBody '⟩'

Tuple ::= '(' ')' | '(' [NEWLINE INDENT] Expr ',' [NEWLINE [INDENT]]
[EnumerationBody] ')'

CollectionBody ::= QuantificationBody | EnumerationBody

EnumerationBody ::= [NEWLINE INDENT] Expr {',' [NEWLINE [INDENT]]
Expr} [NEWLINE DEDENT]

CompoundStmt ::= IfStmt | ForStmt | WhileStmt
| RecordStmt | ProcedureStmt

IfStmt ::= 'if' Predicate ':' Block [ElseStmt]

ElseStmt ::= 'else :' Block
| 'else if' Predicate ':' Block [ElseStmt]

ForStmt ::= 'for' IdentList '∈' Expr ':' Block

WhileStmt ::= 'while' Expr ':' Block

RecordStmt ::= 'record' IDENTIFIER ':' NEWLINE INDENT (
'skip' | TypedName {(',' [NEWLINE] | NEWLINE) TypedName }
) NEWLINE DEDENT

ProcedureStmt ::= 'procedure' IDENTIFIER '(' TypedName {',' TypedName
} ')' '→' Expr ':' Block

TypedName ::= IDENTIFIER [':' Expr]

Block ::= SimpleStmt
| NEWLINE INDENT Statements DEDENT

3.3 Static Analysis - Type System

Let Γ be the type environment set consisting of elements either $identifier$: $Type$ or $Type$. The typing rules have been organized in distinct categories for readability. The notation $Type[x, y, z]$ denotes refinements on $Type$ either through instantiating parametric (generic) types or restricting values by way of `with` clauses.

Type refinements may be applicable to types with certain traits (ex., min/-max values for ordered types, definedness of expressions with certain values, set sizes).

3.3.1 Base Environment Interactions

Includes: Mechanisms for adding/extracting variables from the type environment.

$$\frac{}{\varnothing \vdash \diamond} \quad \text{(Env } \varnothing\text{)}$$

$$\frac{\Gamma \vdash A \quad I \notin dom(\Gamma)}{\Gamma, I : A \vdash \diamond} \quad \text{(Env } I\text{)}$$

$$\frac{\Gamma_1, I : A, \Gamma_2 \vdash \diamond}{\Gamma_1, I : A, \Gamma_2 \vdash I : A} \quad \text{(Fetch Identifier)}$$

3.3.2 The Set Type

Includes: Universal type; everything is a set. The empty set is a part of every type (aside from primitives).

$$\frac{\Gamma \vdash T}{\Gamma \vdash set[T]} \quad \text{(Powerset)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash EmptySet} \quad \text{(Emptyset)}$$

$$\frac{\Gamma \vdash EmptySet, set[T]}{EmptySet \leq set[T]} \quad \text{(Emptyset Top)}$$

$$\frac{\Gamma \vdash e_1, ..., e_n : A}{\Gamma \vdash \{e_1, ..., e_n\} : set[A]} \quad \text{(Set Enumeration)}$$

3.3.3 Primitive Types

Includes: Basic types built in to the language, like numbers (int and float), strings, and booleans. Although these could all be represented as sets, it is useful to distinguish these common types from other collections or user-defined types.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash bool} \quad \text{(Primitives - bool)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash int} \quad \text{(Primitives - int)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash float} \quad \text{(Primitives - float)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash str} \quad \text{(Primitives - str)}$$

$$\frac{\Gamma \vdash int}{\Gamma \vdash nat \quad nat = \{\, n \geq 0 \cdot n \in int \,\}} \quad \text{(Nat from int)}$$

$$\frac{\Gamma \vdash int, float}{int \leq float} \quad \text{(Int from float)}$$

3.3.4 Collection Types

Includes: Sequences, relations, enums (frozen sets), and bags (multisets); Syntactic sugar for abstract data types defined in terms of sets.

$$\frac{\Gamma \vdash T_1, ..., T_n}{\Gamma \vdash tuple[T_1, ..., T_n] \quad tuple[T_1, ..., T_n] = T_1 \times ... \times T_n} \quad \text{(Tuples)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash () : EmptyTuple} \quad \text{(Empty Tuple)}$$

$$\frac{\Gamma \vdash EmptyTuple, T}{\Gamma \vdash EmptyTuple \subseteq tuple[T]} \quad \text{(Empty Tuple Top)}$$

$$\frac{\Gamma \vdash e_1 : A_1, ..., e_n : A_n}{\Gamma \vdash (e_1, ..., e_n) : tuple[A_1, ..., A_n]} \quad \text{(Tuple Enumeration)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \leftrightarrow T_2 \quad T_1 \leftrightarrow T_2 = set[tuple[T_1, T_2]]} \quad \text{(Relation Type)}$$

$$\frac{\Gamma \vdash T, nat}{\Gamma \vdash bag[T] = T \nrightarrow nat} \quad \text{(Bag Type)}$$

$$\frac{\Gamma \vdash e_1, ..., e_n : A}{\Gamma \vdash [\![e_1, ..., e_n]\!] : bag[A]} \quad \text{(Bag Enumeration)}$$

$$\frac{\Gamma \vdash T, nat}{\Gamma \vdash bag[T] = nat \nrightarrow T} \quad \text{(Sequence Type)}$$

$$\frac{\Gamma \vdash e_1, ..., e_n : A}{\Gamma \vdash \langle e_1, ..., e_n \rangle : sequence[A]} \quad \text{(Sequence Enumeration)}$$

$$\frac{\Gamma \vdash s_1, ..., s_n : str \qquad A = \{s_1, ..., s_n\} \qquad immutable(A)}{\Gamma \vdash enum[A]} \quad \text{(Enum from static set)}$$

3.3.5 Relation Subtypes

Includes: Relation subtypes examine four key properties of relations: totality (t), surjectivity (t_r), one-to-manyness ($1-$), and many-to-oneness ($1-_r$). Syntactic sugar to produce refined relations with these properties follows the definitions presented in Event-B. See Section 5.1.3 for more details.

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \trel T_2 \quad T_1 \trel T_2 = (T_1 \leftrightarrow T_2)[\text{with } t]} \quad \text{(Relation Subtype - Total Relation)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \srel T_2 \quad T_1 \srel T_2 = (T_1 \leftrightarrow T_2)[\text{with } t_r]} \quad \text{(Relation Subtype - Surjective Relation)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \strel T_2 \quad T_1 \strel T_2 = (T_1 \leftrightarrow T_2)[\text{with } t,\ \text{with } t_r]} \quad \text{(Relation Subtype - Total Surjective Relation)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \pfun T_2 \quad T_1 \pfun T_2 = (T_1 \leftrightarrow T_2)[\text{with } 1-]} \quad \text{(Relation Subtype - Partial Function)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \rightarrow T_2 \quad T_1 \rightarrow T_2 = (T_1 \leftrightarrow T_2)[\text{with } t,\ \text{with } 1-]} \quad \text{(Relation Subtype - Total Function)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \pinj T_2 \quad T_1 \pinj T_2 = (T_1 \leftrightarrow T_2)[\text{with } 1-,\ \text{with } 1-_r]} \quad \text{(Relation Subtype - Partial Injection)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \rightarrowtail T_2 \quad T_1 \rightarrowtail T_2 = (T_1 \leftrightarrow T_2)[\text{with } t,\ \text{with } 1-,\ \text{with } 1-_r]} \quad \text{(Relation Subtype - Total Injection)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \psur T_2 \quad T_1 \psur T_2 = (T_1 \leftrightarrow T_2)[\text{with } t_r,\ \text{with } 1-]} \quad \text{(Relation Subtype - Partial Surjection)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \twoheadrightarrow T_2 \quad T_1 \twoheadrightarrow T_2 = (T_1 \leftrightarrow T_2)[\text{with } t,\ \text{with } t_r,\ \text{with } 1-]} \quad \text{(Relation Subtype - Total Surjection)}$$

$$\frac{\Gamma \vdash set[T_1], set[T_2]}{\Gamma \vdash T_1 \tbij T_2 \quad T_1 \tbij T_2 = (T_1 \leftrightarrow T_2)[\text{with } t,\ \text{with } t_r,\ \text{with } 1-,\ \text{with } 1-_r]} \quad \text{(Relation Subtype - Bijection)}$$

3.3.6 Simple Commands

Includes: Assignment, type assignment, type refinements, simple programming language commands/statements.

$$\frac{\Gamma \vdash E : A \qquad I \notin dom(\Gamma) \vee \Gamma \vdash I : A}{\Gamma \vdash I : A := E} \quad \text{(Variable Assignment)}$$

$$\frac{\Gamma \vdash T \qquad I \notin dom(\Gamma)}{\Gamma \vdash I := T} \quad \text{(Type Alias Assignment)}$$

$$\frac{\Gamma \vdash I := T}{\Gamma, I \vdash \diamond \quad I = T} \quad \text{(Type Alias)}$$

$$\frac{\Gamma \vdash E : A, E_1 : A_1, ..., E_n : A_n \qquad (I \notin dom(\Gamma) \vee \Gamma \vdash I : A)}{\Gamma \vdash I : A[\text{with } E_1 : A_1, ..., \text{ with } E_n : A_n] := E} \quad \text{(Refined Variable Assignment)}$$

$$\frac{\Gamma \vdash T, E_1 : A_1, ..., E_n : A_n \qquad I \notin dom(\Gamma)}{\Gamma \vdash I := T[\text{with } E_1 : A_1, ..., \text{ with } E_n : A_n]} \quad \text{(Refined Type Alias)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{break} : C} \quad \text{(Command - break)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{continue} : C} \quad \text{(Command - continue)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{skip} : C} \quad \text{(Command - skip)}$$

$$\frac{\Gamma \vdash E : A}{\Gamma \vdash (\text{return } E) : C} \quad \text{(Command - return)}$$

3.3.7 Quantification Expressions

Includes: Set, sequence, relation, and bag comprehensions, in addition to (set based) lambda expressions. Comprehension variables may be bound through either an $Identifier$ or an arbitrarily nested tuple of $Identifiers$

$$\frac{\Gamma \vdash E : A, P : bool \qquad I_1 : T_1, ..., I_n : T_n \text{ are vars in } E}{\Gamma \vdash (\lambda I_1, ..., I_n \cdot P \mid E) : (T_1, ..., T_n) \to A} \quad \text{(Lambda Expression)}$$

$$\frac{\begin{array}{r}\Gamma \vdash E : A, P : bool \\ I_1, ..., I_n : TupleIdentifier \\ Ids = [\![identifiers(I_i) \cdot 0 \leq i \leq n]\!] \\ \forall e \mapsto count \in Ids \cdot count = 1 \wedge e \notin dom(\Gamma) \\ M = \{I_i \mapsto T \cdot 0 \leq i \leq n \\ \wedge \exists g \cdot \Gamma \vdash g : set[T] \\ \wedge structuralMatch(e, T) \\ \wedge hasGenerator(P, e \in g) \\ \wedge structuralMatchInAllORs(e, T, P)\}\end{array}}{\Gamma \vdash (I_1, ..., I_n \cdot P \mid E) : QuantificationBody[M, A]} \quad \text{(Quantification Body [4])}$$

$$\frac{\Gamma \vdash (I \cdot P \mid E) : QuantificationBody[M, A]}{\Gamma \vdash (\bigcup I \cdot P \mid E) : A} \quad \text{(General Union)}$$

$$\frac{\Gamma \vdash (I \cdot P \mid E) : QuantificationBody[M, A]}{\Gamma \vdash (\bigcap I \cdot P \mid E) : A} \quad \text{(General Intersection)}$$

$$\frac{\Gamma \vdash (I \cdot P) : QuantificationBody[M, bool]}{\Gamma \vdash (\forall I \cdot P) : bool} \quad \text{(Forall)}$$

$$\frac{\Gamma \vdash (I \cdot P) : QuantificationBody[M, bool]}{\Gamma \vdash (\exists I \cdot P) : bool} \quad \text{(Exists)}$$

$$\frac{\Gamma \vdash (I \cdot P \mid E) : QuantificationBody[M, A]}{\Gamma \vdash \{\, I \cdot P \mid E \,\} : set[A]} \quad \text{(Set Comprehension)}$$

$$\frac{\Gamma \vdash (I \cdot P \mid E) : QuantificationBody[M, A]}{\Gamma \vdash [\![\, I \cdot P \mid E \,]\!] : bag[A]} \quad \text{(Bag Comprehension)}$$

$$\frac{\Gamma \vdash (I \cdot P \mid E) : QuantificationBody[M, A]}{\Gamma \vdash [\, I \cdot P \mid E \,] : sequence[A]} \quad \text{(Sequence Comprehension)}$$

3.3.8 Boolean Operations

Includes: Common predicate logic operations (equivalence, implication, and, or, etc.).

$$\frac{\Gamma \vdash b_1, b_2 : bool}{\Gamma \vdash b_1 \equiv b_2 : bool} \quad \text{(Boolean Operations - Equivalence)}$$

$$\frac{\Gamma \vdash b_1, b_2 : bool}{\Gamma \vdash b_1 \not\equiv b_2 : bool} \quad \text{(Boolean Operations - Not Equivalence)}$$

$$\frac{\Gamma \vdash b_1, b_2 : bool}{\Gamma \vdash b_1 \implies b_2 : bool} \quad \text{(Boolean Operations - Implication)}$$

$$\frac{\Gamma \vdash b_1, b_2 : bool}{\Gamma \vdash b_1 \wedge b_2 : bool} \quad \text{(Boolean Operations - And)}$$

$$\frac{\Gamma \vdash b_1, b_2 : bool}{\Gamma \vdash b_1 \vee b_2 : bool} \quad \text{(Boolean Operations - Or)}$$

$$\frac{\Gamma \vdash b_1 : bool}{\Gamma \vdash \neg b_1 : bool} \quad \text{(Boolean Operations - Negation)}$$

3.3.9 Equality and Membership

Includes: Equality and ordering comparisons, set membership.

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2}{\Gamma \vdash a_1 = a_2 : bool} \quad \text{(Equals)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2}{\Gamma \vdash a_1 \neq a_2 : bool} \quad \text{(Not Equals)}$$

$$\frac{\Gamma \vdash a_1, a_2 : T \quad T \in \{int, float, nat\}}{\Gamma \vdash a_1 < a_2 : bool} \quad \text{(Less Than)}$$

$$\frac{\Gamma \vdash a_1, a_2 : T \quad T \in \{int, float, nat\}}{\Gamma \vdash a_1 \leq a_2 : bool} \quad \text{(Less Than or Equal)}$$

$$\frac{\Gamma \vdash a_1, a_2 : T \quad T \in \{int, float, nat\}}{\Gamma \vdash a_1 > a_2 : bool} \quad \text{(Greater Than)}$$

$$\frac{\Gamma \vdash a_1, a_2 : T \quad T \in \{int, float, nat\}}{\Gamma \vdash a_1 \geq a_2 : bool} \quad \text{(Greater Than or Equal)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : set[T_2]}{\Gamma \vdash a_1 \in a_2 : bool} \quad \text{(Set Membership)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : set[T_2]}{\Gamma \vdash a_1 \notin a_2 : bool} \quad \text{(Not Set Membership)}$$

3.3.10 Set Operations

Includes: Subset comparisons, union, intersection, cartesian product, etc. Also includes Maplet and Range.

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \subset a_2 : bool} \quad \text{(Set Operations - Subset)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \subseteq a_2 : bool} \quad \text{(Set Operations - Subset Equal)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \supset a_2 : bool} \quad \text{(Set Operations - Superset)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \supseteq a_2 : bool} \quad \text{(Set Operations - Superset Equal)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \not\subset a_2 : bool} \quad \text{(Set Operations - Not Subset)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \not\subseteq a_2 : bool} \quad \text{(Set Operations - Not Subset Equal)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \not\supset a_2 : bool} \quad \text{(Set Operations - Not Superset)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \not\supseteq a_2 : bool} \quad \text{(Set Operations - Not Superset Equal)}$$

$$\frac{\Gamma \vdash a_1, a_2 : set[T]}{\Gamma \vdash a_1 \cup a_2 : set[T]} \quad \text{(Set Operations - Union)}$$

$$\frac{\Gamma \vdash a_1, a_2 : set[T]}{\Gamma \vdash a_1 \cap a_2 : set[T]} \quad \text{(Set Operations - Intersection)}$$

$$\frac{\Gamma \vdash a_1, a_2 : set[T]}{\Gamma \vdash a_1 \setminus a_2 : set[T]} \quad \text{(Set Operations - Difference)}$$

$$\frac{\Gamma \vdash a_1 : set[T_1], a_2 : set[T_2]}{\Gamma \vdash a_1 \times a_2 : T_1 \leftrightarrow T_2} \quad \text{(Cartesian Product)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2}{\Gamma \vdash a_1 \mapsto a_2 : tuple[T_1, T_2]} \quad \text{(Maplet)}$$

$$\frac{\Gamma \vdash a_1, a_2 : int}{\Gamma \vdash a_1 \mathinner{..} a_2 : set[int, \text{ with } exclude(n) = a_1 < n, \text{ with } exclude(n) = a_2 > n]} \quad \text{(Numerical Range)}$$

$$\frac{\Gamma \vdash a : T}{\Gamma \vdash \mathbb{P}(a) : set[T]} \quad \text{(Set Operations - Powerset)}$$

3.3.11 Bag Operations

Includes: Bag union, bag intersection, etc.

$$\frac{\Gamma \vdash a_1, a_2 : bag[T]}{\Gamma \vdash a_1 \cup a_2 : bag[T]} \quad \text{(Bag Operations - (Max) Union)}$$

$$\frac{\Gamma \vdash a_1, a_2 : bag[T]}{\Gamma \vdash a_1 \cap a_2 : bag[T]} \quad \text{(Bag Operations - (Min) Intersection)}$$

$$\frac{\Gamma \vdash a_1, a_2 : bag[T]}{\Gamma \vdash a_1 + a_2 : bag[T]} \quad \text{(Bag Operations - Addition)}$$

$$\frac{\Gamma \vdash a_1, a_2 : bag[T]}{\Gamma \vdash a_1 - a_2 : bag[T]} \quad \text{(Bag Operations - Subtraction)}$$

$$\frac{\Gamma \vdash a : T_1 \leftrightarrow T_2, b : bag[T_1]}{\Gamma \vdash a[b] : bag[T_2] \quad a[b] = a^{-1} \circ b} \quad \text{(Bag Operations - Image)}$$

3.3.12 Relation Operations

Includes: Relation overriding, composition, domain/range manipulation. See Section 5.1.3 for changes in relational subtypes after applying operations.

$$\frac{\Gamma \vdash a : T_1 \leftrightarrow T_2, b : T_1}{\Gamma \vdash a(b) : T_2} \quad \text{(Relation Operations - Function Call)}$$

$$\frac{\Gamma \vdash a : T_1 \leftrightarrow T_2, b : set[T_1]}{\Gamma \vdash a[b] : set[T_2]} \quad \text{(Relation Operations - Image)}$$

$$\frac{\Gamma \vdash a_1, a_2 : T_1 \leftrightarrow T_2}{\Gamma \vdash a_1 \oplus a_2 : T_1 \leftrightarrow T_2} \quad \text{(Relation Operations - Overriding)}$$

$$\frac{\Gamma \vdash a_1 : T_1 \leftrightarrow T_2, a_2 : T_2 \leftrightarrow T_3}{\Gamma \vdash a_1 \circ a_2 : T_1 \leftrightarrow T_3} \quad \text{(Relation Operations - Composition)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_1 \leftrightarrow T_2}{\Gamma \vdash a_1 \lhd a_2 : T_1 \leftrightarrow T_2} \quad \text{(Relation Operations - Domain Restriction)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_1 \leftrightarrow T_2}{\Gamma \vdash a_1 \lhd\!\!\!- a_2 : T_1 \leftrightarrow T_2} \quad \text{(Relation Operations - Domain Subtraction)}$$

$$\frac{\Gamma \vdash a_1 : T_1 \leftrightarrow T_2, a_2 : T_2}{\Gamma \vdash a_1 \rhd a_2 : T_1 \leftrightarrow T_2} \quad \text{(Relation Operations - Range Restriction)}$$

$$\frac{\Gamma \vdash a_1 : T_1 \leftrightarrow T_2, a_2 : T_2}{\Gamma \vdash a_1 \rhd\!\!\!- a_2 : T_1 \leftrightarrow T_2} \quad \text{(Relation Operations - Range Subtraction)}$$

3.3.13 Sequence Operations

Includes: Concatenation

$$\frac{\Gamma \vdash a_1, a_2 : sequence[T]}{\Gamma \vdash a_1 +\!\!+ a_2 : sequence[T]} \quad \text{(Sequence Operations - Concatenation)}$$

3.3.14 Numerical Operations

Includes: Addition, subtraction, multiplication, etc. on natrual numbers, integers, and floats.

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat\}}{\Gamma \vdash a_1 \text{ div } a_2 : max(T_1, T_2)} \quad \text{(Integer Operations - Division)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat\}}{\Gamma \vdash a_1 \text{ mod } a_2 : max(T_1, T_2)} \quad \text{(Integer Operations - Modulo)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat, float\}}{\Gamma \vdash a_1 + a_2 : max(T_1, T_2)} \quad \text{(Numerical Operations - Addition)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat, float\}}{\Gamma \vdash a_1 - a_2 : max(T_1, T_2, int)} \quad \text{(Numerical Operations - Subtraction)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat, float\}}{\Gamma \vdash a_1 * a_2 : float} \quad \text{(Numerical Operations - Floating Division)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat, float\}}{\Gamma \vdash a_1 \text{ div } a_2 : max(T_1, T_2)} \quad \text{(Numerical Operations - Multiplication)}$$

$$\frac{\Gamma \vdash a_1 : T \quad T \in \{int, nat, float\}}{\Gamma \vdash -a_1 : max(int, T)} \quad \text{(Numerical Operations - Negation)}$$

$$\frac{\Gamma \vdash a_1 : T_1, a_2 : T_2 \quad T_i \in \{int, nat, float\}}{\Gamma \vdash a_1 \hat{} \ a_2 : max(T_1, T_2)} \quad \text{(Numerical Operations - Exponentiation)}$$

3.3.15 Record Types

Includes: Types for named tuples.

$$\frac{\Gamma \vdash a : Record[I : A, ...]}{\Gamma \vdash a.I : A} \quad \text{(Records - Access)}$$

$$\frac{\Gamma \vdash A_1, ..., A_n \quad \forall I_i, I_j \cdot I_i \neq I_j}{\Gamma \vdash Record[I_1 : A_1, ..., I_n : A_n]} \quad \text{(Records - Type Definition)}$$

$$\frac{\Gamma \vdash a_1 : A_1, ..., a_n : A_n}{\Gamma \vdash record(a_1 = A_1, ..., a_n = A_n) : Record[a_1 : A_1, ..., a_n : A_n]} \quad \text{(Records - Initialization)}$$

3.3.16 Compound Commands

Includes: If, While, For, and Import statements, along with Procedure definitions and calls.

$$\frac{\Gamma \vdash C_1, C_2}{\Gamma \vdash C_1 \text{ \textbackslash n } C_2} \quad \text{(Command - Composition)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{import } M : C} \quad \text{(Command - Valid Import Module)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{from } M \text{ import } I_1, ..., I_n : C} \quad \text{(Command - Valid Import Names)}$$

$$\frac{\Gamma \vdash \text{import } M \text{ (with identifiers and types } I_1 : A_1, ..., I_n : A_n)}{\Gamma, M = record(I_1 = A_1, ..., I_n = A_n) \vdash \diamond} \quad \text{(Command - Import Module)}$$

$$\frac{\Gamma \vdash \text{from } M \text{ import } I_1, ..., I_n \text{ (with types } A_1, ..., A_n)}{\Gamma, I_1 : A_1, ..., I_n : A_n \vdash \diamond} \quad \text{(Command - Import Names)}$$

$$\frac{\Gamma \vdash C, E, A_1, ..., A_n}{\Gamma \vdash \text{ procedure } I = C \therefore I : procedure[A_1, ..., A_n, E]} \quad \text{(Command - Procedure Definition)}$$

$$\frac{\Gamma \vdash procedure[A_1, ..., A_n, E], a_1 : A_1, ..., a_n : A_n}{\Gamma \vdash I(a_1, ..., a_n) : E} \quad \text{(Command - Procedure Call)}$$

$$\frac{\Gamma \vdash C, b : bool}{\Gamma \vdash \text{if } b : C} \quad \text{(Command - If)}$$

$$\frac{\Gamma \vdash C_1, C_2, b : bool}{\Gamma \vdash \text{if } b : C_1 \text{ else} : C_2} \quad \text{(Command - If Else)}$$

$$\frac{\Gamma \vdash C, b : bool}{\Gamma \vdash \text{while } b : C} \quad \text{(Command - While)}$$

$$\frac{\Gamma \vdash C, p : bool \quad p = i \in G}{\Gamma \vdash \text{for } p : C} \quad \text{(Command - For)}$$

$$\frac{\Gamma \vdash D \therefore I : A \quad \Gamma, I : A \vdash C}{\Gamma \vdash C \text{ substitute vars in } D} \quad \text{(Command - Block)}$$

3.3.17 Relational Subtypes After Operations

Includes: Changes in relational subtyping (totality, surjectivity, etc.) after applying operations. For simplicity, we implement subtype changes in a few binary operators as a boolean mask representing: [with total (t), with total on range (t_r), with one-to-manyness ($1-$), with many-to-oneness ($1-_r$)]. Subtyping is done conservatively: we keep subtype properties only when they are guaranteed. See Section 5.1.3 for more on relational subtyping.

$$\frac{\Gamma \vdash s : set[T_1], r : (T_1 \leftrightarrow T_2)[True, x, y, z]}{\Gamma \vdash (s \lhd r) : (T_1 \leftrightarrow T_2)[False, x, y, z]}$$ (Relational Subtype - Domain Restriction)

$$\frac{\Gamma \vdash s : set[T_1], r : (T_1 \leftrightarrow T_2)[True, x, y, z]}{\Gamma \vdash (s \lhd\!\!\!- r) : (T_1 \leftrightarrow T_2)[False, x, y, z]}$$ (Relational Subtype - Domain Subtraction)

$$\frac{\Gamma \vdash s : set[T_2], r : (T_1 \leftrightarrow T_2)[x, True, y, z]}{\Gamma \vdash (r \rhd s) : (T_1 \leftrightarrow T_2)[x, False, y, z]}$$ (Relational Subtype - Range Restriction)

$$\frac{\Gamma \vdash s : set[T_2], r : (T_1 \leftrightarrow T_2)[x, True, y, z]}{\Gamma \vdash (r \rhd\!\!\!- s) : (T_1 \leftrightarrow T_2)[x, False, y, z]}$$ (Relational Subtype - Range Subtraction)

$$\frac{\Gamma \vdash r : (T_1 \leftrightarrow T_2)[w, x, y, z]}{\Gamma \vdash r^{-1} : (T_1 \leftrightarrow T_2)[x, w, z, y]}$$ (Relational Subtype - Inverse)

$$\frac{\Gamma \vdash r : (T_1 \leftrightarrow T_2)[a, b, c, d], t : (T_1 \leftrightarrow T_2)[w, x, y, z]}{\Gamma \vdash (r \oplus t) : (T_1 \leftrightarrow T_2)[w \le a \wedge c \le y, d \wedge z]}$$ (Relational Subtype - Overriding)

$$\frac{\Gamma \vdash r : (T_1 \leftrightarrow T_2)[a, b, c, d], t : (T_2 \leftrightarrow T_3)[w, x, y, z]}{\Gamma \vdash (r \circ t) : (T_1 \leftrightarrow T_3)[a, b \wedge x, c \wedge y, d \wedge z]}$$ (Relational Subtype - Composition)

3.3.18 Built-in Functions

Includes: Common set-related functions.

$$\frac{\Gamma \vdash T[\text{with } order \in traits(T)]}{\Gamma \vdash min : set[T] \to T}$$ (Built-in - Minimum)

$$\frac{\Gamma \vdash T, int}{\Gamma \vdash min : set[T] \times (T \to int) \to T}$$ (Built-in - Mapped Minimum)

$$\frac{\Gamma \vdash T[\text{with } order \in traits(T)]}{\Gamma \vdash max : set[T] \to T}$$ (Built-in - Maximum)

$$\frac{\Gamma \vdash T, int}{\Gamma \vdash max : set[T] \times (T \to int) \to T}$$ (Built-in - Mapped Maximum)

$$\frac{\Gamma \vdash T}{\Gamma \vdash choice : set[T] \to T}$$ (Built-in - Choice)

$$\frac{\Gamma \vdash T}{\Gamma \vdash dom : T_1 \leftrightarrow T_2 \to set[T_1]}$$ (Built-in - Domain)

$$\frac{\Gamma \vdash T}{\Gamma \vdash ran : T_1 \leftrightarrow T_2 \to set[T_2]}$$ (Built-in - Range)

$$\frac{\Gamma \vdash T}{\Gamma \vdash card : set[T] \to nat}$$ (Built-in - Cardinality)

$$\frac{\Gamma \vdash T}{\Gamma \vdash size : bag[T] \to nat} \quad \text{(Built-in - Bag Size)}$$

$$\frac{\Gamma \vdash T \qquad T \in \{int, float, nat\}}{\Gamma \vdash sum : set[T] \to T} \quad \text{(Built-in - Sum)}$$

3.4 Definedness

3.5 Language Examples

4 Features and Behaviours

5 User-Facing Specification

5.1 (Abstract) Data Types

For the most part, Simile's semantics resemble those of set theory and specification languages (specifically, Event-B). Aside from a few primitive types that abstract over bare set theory and programming language specific structures, like integer arithmetic and procedure definitions, all objects inherit properties from sets. For example, a sequence can be modelled as a set of pairs from natural numbers to the sequence element type.

In the forthcoming sections, we explicitly state the design of these set-theory-inspired types, useful properties, and underlying implementations.

5.1.1 Primitives

With consideration to execution efficiency and the extensive amount of existing optimization in modern languages, Simile makes use of the following basic types:

Booleans As the cornerstone of logical reasoning, booleans deserve an explicit type with reserved identifiers: $\mathbb{B} = \{True, False\}$. Further, Simile provides a healthy supply of common operations beyond $\wedge$ and $\vee$: $\Longrightarrow, \equiv, \Longleftarrow, \forall, \exists$, etc.

Integers While integers are a very mature type in the world of computing, trade-offs between arbitrary-precision and fixed-point arithmetic may require testing and careful review.

If Simile chooses to represent numbers through conventional fixed point arithmetic, struggles with overflow and extremely large numbers are compounded by the innate non-determinism within Simile. For example, calculating the sum of a set of very large numbers can be done in arbitrary order, according to the semantics of mathematics (and Simile). But in fixed-point arithmetic, the resulting value can very well differ based on the order of operations. In other words, commutative math operations are no longer commutative under fixed precision. At some point in the computation pipeline, Simile will need to decide the order in which to evaluate the set, which may cause hard-to-spot bugs and unexpected runtime

failures.

Conversely, choosing arbitrary precision integers may bring Simile operators closer to mathematics at the cost of speed. The core features of Simile, while targeting formal methods users, attempt to efficiently execute set and relation operations. Adding checks for overflow on every mathematical operation may prove detrimental to performance.

The implementation decision essentially is one of context vs. safety. To benchmark Simile with these types, we gather a list of examples and judge whether the performance hit is worth the reliability benefits. Specific questions to examine: How often does overflow occur in these examples? Do numbers approach the upper limits of fixed precision often enough to warrant accommodating infrastructure? What is the performance of fixed vs. arbitrary precision calculations.

Floating Point Numbers In a similar vein to integers, floats are an imperfect translation of real numbers into efficiently calculable representations. Since floats (or even reals) are not used as often in specifications as integers, we leave the implementation as the conventional 64-bit representation. However, we note that non-determinism on these operations will prove especially tricky, since the result of commutative complements may differ without explicit over/underflow.

Strings Strings are syntactic sugar over Sequences of characters, represented internally as a sequence of (fixed precision) integers. Eventually, Simile aims to support built-in string operations as seen in many common programming languages.

5.1.2 Sets

Foundational Simile type: an unordered, unique collection of objects.

Sets may be used in two ways:

1. Enumerations by way of $\{x, y, z\}$ (read: The set of elements x, y, and z)
2. Comprehensions of the form $\{ x \cdot x \in S \mid f(x) \}$ (read: The set of function f applied to all elements in S). A small effort is made to accommodate a shorthand form, wherein $\{ x \cdot x \in S \mid x \}$ may be written as $\{ x \cdot x \in S \}$, but this syntax easily confuses bound/unbound variables and is not recommended for complex comprehension expressions.

Sets, as mathematical objects with only two major restrictions, lend themselves well to a variety of computational implementations. Arrays, ordered arrays, hash maps, bit sets, roaring bit sets, path maps, and tries can all be used to satisfy the uniqueness property and hide an internally ordered representation from users.

No matter the underlying implementation, the optimizer may make use of syntactically-analyzed set sizes to direct the choice of lowered iterator. Further, recording the cardinality of a set at runtime may prove beneficial to the

performance of common cardinality-based specification expressions at a very small memory cost. Other properties based on element type, like max/min for numeric types, limits for numeric ranges, maximum sizes, may be useful.

5.1.3 Relations

Relations are sets that make use of paired/maplet elements (i.e., they define a relationship between two sets). Like sets, implementations may vary greatly, with the potential to increase efficiency of relation-specific operations.

Like sets, relations may benefit from compile-time computed properties: size of the domain, size of the range, totality, many-to-oneness, $O(1)$ access to the domain and range, and access to associated domain/range properties.

Relational Subtypes The notion of a relation is a broad classification of a set of pairs, where the domain, range, and relationship are not bound by any conditions. However, when generating code based on purely mathematical relations, it can be advantageous to use notions of totality, injectivity, surjectivity, etc. In particular, these named relational subtypes can be described with four properties: total on the domain, total on the range, one-to-many, and many-to-one.

The below table converts conventional notation into these four subtypes:

Relational Subtype	Properties			
	T	TR	1-	1-R
Relation				
				✓
Partial Function			✓	
Partial Injection			✓	✓
Surjective Relation		✓		
		✓		✓
Partial Surjection		✓	✓	
		✓	✓	✓
Total Relation	✓			
	✓			✓
Total Function	✓		✓	
Total Injection	✓		✓	✓
Total Surjective Relation	✓	✓		
	✓	✓		✓
Total Surjection	✓	✓	✓	
Bijection	✓	✓	✓	✓

Table 3: Conventional mathematical notation of relational subtypes decomposed into four properties: domain totality (T), range totality (TR), domain one-to-manyness (1-), and range one-to-manyness (1-R)

Subtype Properties Properties that can affect code generation may involve:

- New properties after applying operations on sets/relations
- Size of result
- Validity of operations
- Deterministic nature of operations
- Super/subset properties
- Relational identities

Below is a list of the impact of these properties.
Domain Totality:

- $R[S] \neq \varnothing$
- $R(S)$ is always valid
- R^{-1} is TR
- $|R| \geq dom(R)$

Range Totality:

- R^{-1} is T
- $|R| \geq ran(R)$

Domain One-to-manyness:

- $R(\{e\})$ is deterministic
- $|R[\{e\}]| = 1$
- $|R[S]| \leq |S|$
- $|R| \leq dom(R)$

Range One-to-manyness:

- $|R| \leq ran(R)$

5.1.4 Bags

Bags (or Multisets) are a refinement of relations wherein, for a bag with element type T, the bag is a relational function from $T \to \mathbb{Z}^+$, denoting the count of items in a collection. Once the count of an item reaches 0, the item is removed from the bag.

Bags must be many-to-one and carry specialized behaviours for conventional relation operations. For example, relational image ($R[B]$) with a bag argument will now return a bag, where the number of occurrences of the resulting item replaces what would otherwise be a regular set of those items.

Additional fields to record the size of the bag (sum of the range) may prove useful.

5.1.5 Sequences

Sequences are yet another refinement of relations where a sequence with element type T represents a relational function from $\mathbb{N} \to T$ that is total on some defined bounds.

The dense nature of sequence domains, along with knowledge of sequence sizes, may open up further data type refinements.

5.2 Data Type Operations

5.2.1 Primitives

5.2.2 Sets

5.2.3 Relations

5.2.4 Bags

5.2.5 Sequences

6 Implementation Specification

6.1 Abstract Data Types

6.2 Atomic Operations

6.3 Optimization Frontend

6.4 Optimization Backends

7 AST Lowering and Optimization Rules

Below is a list of rewrite rules for key abstract data types, syntactic sugar, and some builtin functions. Phases are intended to be executed in order; the post-condition of one phase serves as the pre-condition for the next.

7.1 Syntactic Sugar for Bags

$$R[B] \rightsquigarrow R^{-1} \circ B \qquad \text{(Bag Image)}$$

$$S \cup T \rightsquigarrow [\![x \cdot x \in dom(S) \cup dom(T) \wedge n = max(S[x] \cup T[x]) \mid x \mapsto n]\!] \qquad \text{(Bag Predicates - Union)}$$

$$S \cap T \rightsquigarrow [\![x \cdot x \in dom(S) \cap dom(T) \wedge n = min(S[x] \cup T[x]) \wedge n > 0 \mid x \mapsto n]\!] \qquad \text{(Bag Predicates - Intersection)}$$

$$S + T \rightsquigarrow [\![x \cdot x \in dom(S) \cup dom(T) \wedge n = sum(S[x] \cup T[x]) \mid x \mapsto n]\!] \qquad \text{(Bag Predicates - Sum)}$$

$$S - T \rightsquigarrow [\![x \cdot x \in dom(S) \wedge pop(S[x]) - popDefault(T[x], 0) \wedge n > 0 \mid x \mapsto n]\!] \qquad \text{(Bag Predicates - Difference)}$$

7.2 Syntactic Sugar for Sequences

$$L + M \rightsquigarrow L \cup \{\, x \mapsto y \cdot x \mapsto y \in M \mid x + max(dom(L)) \mapsto y \,\} \quad \text{(Concatenation)}$$

$$flatten(X) \rightsquigarrow foldL(\mathbin{+\!\!+}, [], X) \quad \text{(Flatten)}$$

$$foldL(f, e, X) \rightsquigarrow foldL(f, f(e, first(X)), tail(X)) \quad \text{(Fold Left Associative)}$$

$$foldL(f, e, []) \rightsquigarrow e \quad \text{(Fold Left Associative - Empty)}$$

$$first(X) \rightsquigarrow X(0) \quad \text{(First)}$$

$$tail(X) \rightsquigarrow \{\, i \mapsto x \cdot i \mapsto x \in X \wedge i \neq 0 \mid i - 1 \mapsto x \,\} \quad \text{(Tail)}$$

$$append(X, y) \rightsquigarrow X \cup \{max(dom(X)) \mapsto y\} \quad \text{(Append)}$$

$$prepend(X, y) \rightsquigarrow \{\, i \mapsto x \cdot i \mapsto x \in X \mid i + 1 \mapsto x \,\} \cup 0 \mapsto y \quad \text{(Prepend)}$$

7.3 Builtin Functions

$$f(x) := E \rightsquigarrow f := f \oplus \{x \mapsto E\} \quad \text{(Functional Override)}$$

$$card(S) \rightsquigarrow \sum x \cdot x \in S \mid 1 \quad \text{(Cardinality)}$$

$$dom(R) \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \mid x \,\} \quad \text{(Domain)}$$

$$ran(R) \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \,\} \quad \text{(Range)}$$

$$R \oplus Q \rightsquigarrow Q \cup (dom(Q) \lhd R) \quad \text{(Override)}$$

$$S \lhd R \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \wedge x \in S \mid x \mapsto y \,\} \quad \text{(Domain Restriction)}$$

$$S \lhd\!\!\!- R \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \wedge x \notin S \mid x \mapsto y \,\} \quad \text{(Domain Subtraction)}$$

$$R \rhd S \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \wedge y \in S \mid x \mapsto y \,\} \quad \text{(Range Restriction)}$$

$$R \rhd\!\!\!- S \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \wedge y \notin S \mid x \mapsto y \,\} \quad \text{(Range Subtraction)}$$

$$size(S) \rightsquigarrow \sum x \mapsto n \in S \cdot n \quad \text{(Bag Size)}$$

7.4 Comprehension Construction

Intuition All set-like variables and literals are decomposed into set comprehensions.

Post-condition

- All terms with a set-like type (relations, bags, sets, sequences, etc.) must be in comprehension form.

$$R(x) \rightsquigarrow pop(R[x]) \quad \text{(Functional Image [a])}$$
$$R[S] \rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \wedge x \in S \mid y \,\} \quad \text{(Image)}$$
$$x \mapsto y \in S \times T \rightsquigarrow x \in S \wedge y \in T \quad \text{(Product)}$$
$$x \mapsto y \in R^{-1} \rightsquigarrow y \mapsto x \in R \quad \text{(Inverse)}$$
$$x \mapsto y \in (Q \circ R) \rightsquigarrow x \mapsto z \in Q \wedge z' \mapsto y \in R \wedge z = z' \quad \text{(Composition)}$$

[a] May fail if R is not total.

$$S \cup T \rightsquigarrow \{\, x \cdot x \in S \vee x \in T \,\} \quad \text{(Predicate Operations - Union)}$$
$$S \cap T \rightsquigarrow \{\, x \cdot x \in S \wedge x \in T \,\} \quad \text{(Predicate Operations - Intersection)}$$
$$S \setminus T \rightsquigarrow \{\, x \cdot x \in S \wedge x \notin T \,\} \quad \text{(Predicate Operations - Difference)}$$
$$x \in \oplus(E \mid P) \rightsquigarrow P \wedge x = E \quad \text{(Membership Collapse [a])}$$

[a] Rule only matches inside the predicate of a quantifier. Explicitly enumerating all matches for all quantuantification types and predicate cases (ANDs, ORs, etc.) would require too much boilerplate. x must be bound by the encasing quantifier.
The $\oplus$ operator represents any quantifier that returns a set-like type (ex. generalized union/intersection, set comprehension, relation comprehension).

7.5 Disjunctive Normal Form

Intuition All quantifier predicates are expanded to DNF (i.e. $\wedge$-operations nested within top-level $\vee$-operations).

Notes All matches of this phase occur only inside quantifier predicates.

Post-condition

- All terms with a set-like type (relations, bags, sets, sequences, etc.) must be in comprehension form.
- Quantifier predicates are in disjunctive normal form - top level or-clauses with inner and-clauses.
- If no $\vee$ operators exist within a quantifier predicate, the predicate must only contain $\wedge$ operators.

$$x_1 \wedge ... \wedge (x_i \wedge x_{i+1}) \wedge ... \rightsquigarrow x_1 \wedge ... \wedge x_i \wedge x_{i+1} \wedge ... \quad \text{(Flatten Nested } \wedge\text{)}$$
$$x_1 \vee ... \vee (x_i \vee x_{i+1}) \vee ... \rightsquigarrow x_1 \vee ... \vee x_i \vee x_{i+1} \vee ... \quad \text{(Flatten Nested } \vee\text{)}$$
$$\neg\neg x \rightsquigarrow x \quad \text{(Double Negation)}$$
$$\neg(x \vee y) \rightsquigarrow \neg x \wedge \neg y \quad \text{(Distribute De Morgan - Or)}$$
$$\neg(x \wedge y) \rightsquigarrow \neg x \vee \neg y \quad \text{(Distribute De Morgan - And)}$$
$$x \wedge (y \vee z) \rightsquigarrow (x \wedge y) \vee (x \wedge z) \quad \text{(Distribute } \wedge \text{ over } \vee\text{)}$$

7.6 Or-wrapping

Intuition Restructuring quantifier predicates with top-level ORs for later rules.

Post-condition

- All terms with a set-like type (relations, bags, sets, sequences, etc.) must be in comprehension form.
- Quantifier predicates are in disjunctive normal form - top level or-clauses with inner and-clauses.

$$\{ E \cdot \bigwedge P_i \} \rightsquigarrow \{ E \cdot \bigvee \bigwedge P_i \} \qquad \text{(Or-wrapping [a])}$$

[a] To simplify the matching process later on, we wrap every top-level AND statement (which is guaranteed to be a ListOp by the dataclass field type definition) with an OR.

7.7 Generator Selection

Intuition All nested ANDs are placed in a tree structure to better suit loop lowering.

Post-condition

- All terms with a set-like type (relations, bags, sets, sequences, etc.) must be in comprehension form.
- Each top-level or-clause within quantification predicates must have one selected 'generator' predicate (GeneratorSelectionPredicate - GSP) of the form $x \in S$ that loops over its bound dummy variable x.
- All conjunctive clauses are wrapped in either a GSP or CombGSP structure.
- Quantifier predicates are in disjunctive normal form (with GSP replacing AND structures) - top level or-clauses with inner and-clauses.

Notes All matches of this phase occur only inside quantifier predicates. GSP is a tuple of form (generator chain, predicates) that can be flattened to a conjunctive clause. A CombGSP is a triple of form (shared generator, disjunctive children predicates/generators, predicates) that can likewise be flattened into a conjunctive clause. We keep the structure of these tuples distinct to ease the rewriting process.

Brainstorming selection heuristics Conditions to consider:

- Prefer composition chains in case of nested loops (ex. $x \mapsto y \in R \wedge y \mapsto z \in Q$, but what about renaming? - $x \mapsto y \in R \wedge y' \mapsto z \in Q \wedge y = y'$). Perhaps this could just be a preference for relations over sets if we see a nested quantifier.
- Choose most common generator among a list of or-clauses (allows for greater simplification later on)
- Choose smallest generator (by set size). This information may not always be accessible (ex. if a set is created by reading values from standard input). Functions may need to choose this on a case-by-case basis (ie. one function call could have two args, with a small set on the left arg and a large set on the right arg. But what if the sizes are reversed later in the code? We've already statically lowered it).
- When should constant folding happen? Equality substitution necessary for this?
- This may need to work with nesting considerations.
- What if we try moving these optimizations to *loop* structures far later in the pipeline?

$$\bigwedge P_i \rightsquigarrow GSP(\bigwedge P_{g_i}, \bigwedge_{P_i \notin P_g} P_i)$$

(GSP Wrapping [a])

$$GSP(x \wedge xs, P) \vee GSP(x \wedge ys, Q) \rightsquigarrow CombGSP(x, GSP(xs, P) \vee GSP(ys, Q))$$

(Nested Generator Selection [b])

[a] GSP is a tuple-like structure, where all entries can be flattened into an AND. The LH term must occur inside a quantifier's predicate - one generator per top-level or-clause. Chained generators may be necessary to allow for nested loop operations (ie. non-top-level clauses may require generators). P_g is a list of chained generators, specific set membership clauses (of form $x \in S$) distinguished from the rest of $\bigwedge P_i$. Currently, the selection of P_g is arbitrary (and thus the rewrite system is not confluent), but heurestics may be added later to choose better generators.

[b] $free$ is a function that returns true when variables in the predicate are defined (either bound by the current generator or bound prior to the current statement)

7.8 GSP to Loop

Intuition Start lowering expressions into imperative-like loops.

Post-condition

- All quantifiers are transformed into *loop* structures.
- *loop* predicates are of the form $x \in S \wedge \bigwedge P_i$.

- All *loop* predicates have an assigned generator of the form $x \in S$.

$$\oplus E \mid P \rightsquigarrow \begin{array}{l} a := identity(\oplus) \\ \textbf{loop } P: \\ \quad a := accumulate(a, E) \end{array} \qquad \text{(Quantifier Generation [a])}$$

[a] $\oplus$ works for any quantifier (but not $\forall$ and $\exists$). The identity and accumulate functions are determined by the realized $\oplus$. For example, if $\oplus = \sum$, the identity is 0 and accumulate is addition.

$$\begin{array}{l} \textbf{loop } \bigvee P_i: \\ \quad \text{body} \end{array} \rightsquigarrow \begin{array}{l} \textbf{loop } P_0 \\ \quad body \\ \textbf{loop } P_1 \wedge \neg P_0 \\ \quad body \\ \textbf{loop } P_2 \wedge \neg \bigvee_{i<2} P_i \\ \quad body \end{array} \qquad \text{(Top-level Or-Loop)}$$

$$\begin{array}{l} \textbf{loop } GSP(g \wedge gs, P) \\ \quad \text{body} \end{array} \rightsquigarrow \begin{array}{l} \textbf{loop} \\ SingleGSP(g, free(P)) \\ \quad \textbf{loop } GSP(gs, bound(P)) \\ \quad\quad \text{body} \end{array} \qquad \text{(Chained GSP Loop [a])}$$

[a] $free$ considers defined variables at this point in time.

$$\begin{array}{l} \textbf{loop } GSP([], P) \\ \quad \text{body} \end{array} \rightsquigarrow \begin{array}{l} \textbf{if } P \textbf{ then} \\ \quad \text{body} \end{array} \qquad \text{(Empty GSP Loop)}$$

$$\begin{array}{l} \textbf{loop } CombGSP(g, gs, P) \\ \quad \text{body} \end{array} \rightsquigarrow \begin{array}{l} \textbf{loop } SingleGSP(g, P) \\ \quad \textbf{loop } gs_0 \\ \quad\quad \text{body} \\ \quad \textbf{loop } gs_1 \\ \quad\quad \text{body} \\ \quad \ldots \end{array} \qquad \text{(Combined GSP Loop)}$$

7.9 Relational Subtyping Loop Simplification

Intuition Simplify unnecessary iteration structures into direct accesses.

Post-condition

- All quantifiers are transformed into *loop* structures.
- *loop* predicates are of the form $x \in S \wedge \bigwedge P_i$.

- All *loop* predicates have an assigned generator of the form $x \in S$.

No change from previous, but the number of loop structures should not increase.

$$x \in dom(R) \rightsquigarrow True \qquad \text{(Total membership elimination } {}^{a}\text{)}$$
$$x \in ran(R) \rightsquigarrow True \qquad \text{(Surjective membership elimination } {}^{b}\text{)}$$

[a] R is total and x satisfies the type of R's domain. Currently unused since it may eliminate some generators - move higher in the list?

[b] R is surjective and x satisfies the type of R's range. Currently unused since it may eliminate some generators - move higher in the list? Or

loop $SingleGSP(x \mapsto y \in R, x = a \wedge P)$
  body
$\rightsquigarrow$
if $a \in dom(R)$ **then**
  loop $SingleGSP(y \in R[a], P[x := a])$
    body
(Concrete Domain Image)

loop $SingleGSP(x \mapsto y \in R, y = a \wedge P)$
  body
$\rightsquigarrow$
if $a \in ran(R)$ **then**
  loop $SingleGSP(x \in R^{-1}[a], P[y := a])$
    body
(Concrete Range Image [a])

[a] Requires efficient lookups for ran, inverse, and image

loop $SingleGSP(y \in R[x], P)$
  body
where $free(x)$ and R is many-to-one
$\rightsquigarrow$
if $P[y := R(a)]$ **then**
  $body[y := R(a)]$
(Single Element Loop)

loop $SingleGSP(x \in \{y\}, P)$
  body
$\rightsquigarrow$
if $P[x := y]$ **then**
  $body[x := y]$
(Singleton Membership Elimination)

7.10 Loop Code Generation

Intuition Eliminate all intermediate loop structures.

Post-condition

- AST is in imperative code style (for loops, if statements, etc).
- All *loop* and quantification constructs have been eliminated.
- Some variables may not be defined.

loop $SingleGSP(P_g, \bigwedge P_i)$
body

$\rightsquigarrow$

if $\bigwedge_{free(P_i)} P_i$ **then**
for P_g **do**
if $\bigwedge_{bound(P_i)} P_i$ **then**
body

(Conjunct Conditional [a])

[a] Function *free* returns clauses in P that contain only free + defined variables. *bound* returns the clauses that contain the bound variable x or undefined variables. P_g is the selected generator.

7.11 Replace and Simplify

Intuition Eliminate all undefined variables (with implicit $\exists$ quantifiers).

Post-condition

- AST is in imperative code style (for loops, if statements, etc).
- All variables are defined.

if $Identifier(x) = E \wedge P$ **then**
body

$\rightsquigarrow$

if $P[x := E]$ **then**
body[x := E]

(Equality Elimination [a])

[a] x is an undefined, unbound variable in the current scope. E is an expression that does not contain x. Simplify resulting booleans.

$$P \wedge True \rightsquigarrow P \quad \text{(Simplify And-true)}$$
$$P \wedge False \rightsquigarrow False \quad \text{(Simplify And-false)}$$
$$P \vee True \rightsquigarrow True \quad \text{(Simplify Or-true)}$$
$$P \vee False \rightsquigarrow P \quad \text{(Simplify Or-false)}$$
$$Statement(Statement(x)) \rightsquigarrow Statement(x) \quad \text{(Flatten Nested Statements)}$$

References

[1] Christophe Métayer and Laurent Voisin. "The Event-B Mathematical Language". 2007.