Complete TRS Specification for Abstract Collection Types

Anthony Hunt

August 18, 2025

Contents

1 Introduction

This document serves as a living specification of the underlying term rewriting system used in the compiler for a modelling-focused programming language.

2 High Level Strategy

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 $\{ x \cdot x \in S \wedge x \in T \}$, 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 $\neg$, $\vee$, and $\wedge$ 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.

Granular Strategy (Sets)

Set Comprehension Construction Break down all qualifying sets into comprehension forms, collapsing and simplifying where needed.

DNF Predicates Revise comprehension predicates to top-level disjunctive normal form. Each or-clause should have at least one feasible generator. Each clause should record a list of candidate generators

Predicate Simplification Remove superfluous dummy variables, group or-clauses that use the exact same generator (ex. $\{ x \cdot x \in S \land x \neq 0 \lor x \in S \land x = 0 \} \to \{ x \cdot x \in S \land (x \neq 0 \lor x = 0) \}$). Clauses should be group-able based on DNF, and generators should be selected and recorded.

Set Code Generation Converts quantifiers into for-loops and if-statements.

3 Supported Operations

Table 1: Summary table: a few operators on sets and relations.

Sets		Relations	
Syntax	Label/Description	Syntax	Label/Description
$set(T)$	Unordered, unique collection	$S \pfun T$	Partial function
$S \leftrightarrow T$	Relation, $set(S \times T)$	$S \rightarrowtail T$	Total injection
$\varnothing$	Empty set	$a \mapsto b$	Pair (relational element)
$\{a, b, ...\}$	Set enumeration	$dom(S)$	Domain
$\{ x \cdot x \in S \mid P \}$	Set comprehension	$ran(S)$	Range
$S \cup T$	Union	$R[S]$	Relational image
$S \cap T$	Intersection	$R \mathbin{\lhd\mkern-14mu-} Q$	Relational overriding
$S \setminus T$	Difference	$R \circ Q$	Relational composition
$S \times T$	Cartesian Product	$S \lhd R$	Domain restriction
$S \subseteq T$	Subset	R^{-1}	Relational inverse

Table 2: Collection of operators on set data types.

Name	Definition
Empty Set	Creates a set with no elements.
Set Enumeration	Literal collection of elements to create a set.
Set Membership	The term $x \in S$ is True if x can be found somewhere in S.
Union	$S \cup T = \{\, x \cdot x \in S \vee x \in T \,\}$
Intersection	$S \cap T = \{\, x \cdot x \in S \wedge x \in T \,\}$
Difference	$S \setminus T = \{\, x \cdot x \in S \mid x \notin T \,\}$
Cartesian Product	$S \times T = \{\, x \mapsto y \cdot x \in S \wedge y \in T \,\}$
Powerset	$\mathbb{P}(S) = \{\, s \cdot s \subseteq S \,\}$
Magnitude	$\#S = \sum_{x \in S} 1$
Subset	$S \subseteq T \equiv \forall x \in S : s \in T$
Strict Subset	$S \subset T \equiv S \subseteq T \wedge S \neq T$
Superset	$S \supseteq T \equiv \forall x \in T : s \in S$
Strict Superset	$S \supset T \equiv S \supseteq T \wedge S \neq T$
Set Mapping	$f * S = \{\, f(x) \cdot x \in S \,\}$
Set Filter	$p \lhd S = \{\, x \cdot x \in S \mid p(x) \,\}$
Set Quantification (Folding)	$\oplus x \cdot x \in S \mid P$
Cardinality	$card(S) = \sum 1 \cdot x \in S$

Table 3: Collection of operators on bag/multiset data types.

Name	Definition
Empty Set	Creates a set with no elements.
Bag Enumeration	Literal collection of elements to create a set (for now, stored as a tuple of elements and number of occurrences).
Bag Membership	The term $x \in S$ is True if S contains one or more occurrences of x.
Union	$S \cup T = \{\!\mid (x, a+b) \cdot (x,a) \in S \wedge (x,b) \in T \mid a,b \geq 0 \mid\!\}$
Intersection	$S \cap T = \{\!\mid (x, min(a,b)) \cdot (x,a) \in S \wedge (x,b) \in T \mid a,b \geq 0 \mid\!\}$
Difference	$S - T = \{\!\mid (x, a-b) \cdot (x,a) \in S \wedge (x,b) \in T \mid a,b \geq 0 \wedge a-b > 0 \mid\!\}$
Bag Mapping	$f * S = \{\!\mid (f(x), r) \cdot (x,r) \in S \mid\!\}$
Bag Filter	$p \lhd S = \{\!\mid (x,r) \cdot (x,r) \in S \mid p(x) \mid\!\}$
Size	$size(S) = \sum r \cdot (x,r) \in S$
Zero Occurrences	$(x,0) \in S \implies x \notin S$

Table 4: Collection of operators on sequence data types.

Name	Definition
Empty List	Creates a list with no elements.
List Enumeration	Literal collection of elements to create a list.
Construction	Alternative form of List Enumeration.
List Membership	The term x `in` S is True if x can be found somewhere in S.
Append	$[s_1, s_2, ..., s_n] + t = [s_1, s_2, ..., s_n, t]$
Concatenate	$[s_1, ..., s_n] +\!\!+ [t_1, ..., t_n] = [s_1, ..., s_n, t_1, ...t_n]$
Length	$\#S = \sum 1 \cdot x \texttt{ in } S$
List Mapping	$f * S = [\, f(x) \cdot x \texttt{ in } S \,]$
List Filter	$p \lhd S = [\, f(x) \cdot x \texttt{ in } S \mid p(x) \,]$
Associative Reduction	$\oplus/[s_1, s_2, ..., s_n] = s_1 \oplus s_2 \oplus ... \oplus s_n$
Right Fold	$\texttt{foldr}(f, e, [s_1, s_2, ..., s_n]) = f(s_1, f(s_2, f(..., f(s_n, e))))$
Left Fold	$\texttt{foldl}(f, e, [s_1, s_2, ..., s_n]) = f(f(f(f(e, s_1), s_2), ...), s_n)$

Table 5: Collection of operators on relation data types.

Name	Definition
Empty Relation	Creates a relation with no elements.
Relation Enumeration	Literal collection of elements to create a relation.
Identity	$id(S) = \{\, x \mapsto x \cdot x \in S \,\}$
Domain	$dom(R) = \{\, x \cdot x \mapsto y \in R \,\}$
Range	$ran(R) = \{\, y \cdot x \mapsto y \in R \,\}$
Relational Image	$R[S] = \{\, y \cdot x \mapsto y \in R \mid x \in S \,\}$
Overriding	$R \mathbin{\lhd\!\!\!-} Q = Q \cup (dom(Q) \mathbin{\lhd\!\!\!-} R)$
(Forward) Composition	$Q \circ R = \{\, x \mapsto z \cdot x \mapsto y \in R \wedge y \mapsto z \in Q \,\}$
Inverse	$R^{-1} = \{\, y \mapsto x \cdot x \mapsto y \in R \,\}$
Domain Restriction	$S \lhd R = \{\, x \mapsto y \cdot x \mapsto y \in R \mid x \in S \,\}$
Domain Subtraction	$S \mathbin{\lhd\!\!\!-} R = \{\, x \mapsto y \cdot x \mapsto y \in R \mid x \notin S \,\}$
Range Restriction	$R \rhd S = \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \in S \,\}$
Range Subtraction	$R \mathbin{-\!\!\!\rhd} S = \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \notin S \,\}$

4 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 6: 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)

4.1 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 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.

5.1 Syntactic Sugar - Bag Image

$$R[B] \rightsquigarrow R^{-1} \circ B \tag{Bag Image}$$

5.2 Builtin Functions

$$
\begin{aligned}
card(S) &\rightsquigarrow \sum x \cdot x \in S \mid 1 && \text{(Cardinality)} \\
dom(R) &\rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \mid x \,\} && \text{(Domain)} \\
ran(R) &\rightsquigarrow \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \,\} && \text{(Range)}
\end{aligned}
$$

5.3 Sets

Let S, T be sets, P, E expressions, and x, e any type.

5.3.1 Set 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.

$$S \cup T \rightsquigarrow \{\, x \cdot x \in S \lor x \in T \,\} \qquad \text{(Predicate Operations - Union)}$$
$$S \cap T \rightsquigarrow \{\, x \cdot x \in S \land x \in T \,\} \qquad \text{(Predicate Operations - Intersection)}$$
$$S \setminus T \rightsquigarrow \{\, x \cdot x \in S \land x \notin T \,\} \qquad \text{(Predicate Operations - Difference)}$$
$$x \in \{e\} \rightsquigarrow x = e \qquad \text{(Singleton Membership }^{a}\text{)}$$
$$x \in \oplus(E \mid P) \rightsquigarrow P \land x = E \qquad \text{(Membership Collapse }^{b}\text{)}$$

[a]Currently unused. We need to be careful to handle the case where x is a free variable.

[b]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).

5.3.2 Disjunctive Normal Form

Intuition All quantifier predicates are expanded to DNF (i.e. $\land$-operations nested within top-level $\lor$-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 $\lor$ operators exist within a quantifier predicate, the predicate must only contain $\land$ operators.

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

5.3.3 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.

5.3.4 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])

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

(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)

5.3.5 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 $x \in S$.

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

(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.

$$\begin{array}{lcl} \textbf{loop } \bigvee P_i : & & \textbf{loop } P_0 \\ \quad \text{body} & \rightsquigarrow & \quad \textit{body} \\ & & \textbf{loop } P_1 \wedge \neg P_0 \\ & & \quad \textit{body} \\ & & \textbf{loop } P_2 \wedge \neg \bigvee_{i<2} P_i \\ & & \quad \textit{body} \end{array} \tag{Top-level Or-Loop}$$

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

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

$$\begin{array}{lcl} \textbf{loop } GSP(g, P) & \rightsquigarrow & \textbf{loop } SingleGSP(g, P) \\ \quad \text{body} & & \quad \text{body} \end{array} \tag{Single GSP Loop}$$

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

5.3.6 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.

$$\begin{array}{l}\textbf{loop } SingleGSP(P_g, \bigwedge P_i) \\ \quad \text{body}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\textbf{if } \bigwedge_{free(P_i)} P_i \textbf{ then} \\ \quad \textbf{for } P_g \textbf{ do} \\ \quad\quad \textbf{if } \bigwedge_{bound(P_i)} P_i \textbf{ then} \\ \quad\quad\quad \text{body}\end{array}$$

(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.

5.3.7 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.

$$\begin{array}{l}\textbf{if } Identifier(x) = E \wedge P \textbf{ then} \\ \quad \text{body}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\textbf{if } P[x := E] \textbf{ then} \\ \quad \text{body[x := E]}\end{array}$$

(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.

$$P \wedge True \rightsquigarrow P \quad \text{(Simplify And-true)}$$

$$P \wedge False \rightsquigarrow False \quad \text{(Simplify And-false)}$$

$$P \vee True \rightsquigarrow True \quad \text{(Simplify Or-true)}$$

$$P \vee False \rightsquigarrow P \quad \text{(Simplify Or-false)}$$

$$Statement(Statement(x)) \rightsquigarrow Statement(x) \quad \text{(Flatten Nested Statements)}$$

5.4 Relations

$$R[S] \to \{\, x \mapsto y \in R \cdot x \in S \mid y \,\} \quad \text{(Image)}$$

$$x \mapsto y \in S \times T \to x \in S \wedge y \in T \quad \text{(Product)}$$

$$x \mapsto y \in R^{-1} \to y \mapsto x \in R \quad \text{(Inverse)}$$

$$x \mapsto y \in (Q \circ R) \to x \mapsto z \in Q \wedge z' \mapsto y \in R \wedge z = z' \quad \text{(Composition)}$$

$$\exists z \cdot x \mapsto z \in Q \wedge z \mapsto y \in R \to x \mapsto z \in Q \wedge z' \mapsto y \in R \wedge z = z' \quad \text{(Exists Elimination)}$$

$$R \mathbin{\lhd\!\!\!-} Q \to Q \cup (dom(Q) \lhd R) \quad \text{(Override)}$$

$$S \lhd R \rightarrow filter(fst \in S, R) \quad \text{(Domain Restriction)}$$
$$S \mathbin{\lhd\!\!\!-} R \rightarrow filter(fst \notin S, R) \quad \text{(Domain Subtraction)}$$
$$R \rhd S \rightarrow filter(snd \in S, R) \quad \text{(Range Restriction)}$$
$$R \mathbin{\rhd\!\!\!-} S \rightarrow filter(snd \notin S, R) \quad \text{(Range Subtraction)}$$

5.5 Bags

Since bags can be interpreted as a set of tuples $(element, repetitions)$, all set operations apply, except for the overriding operations below.

$$S \cup T \rightarrow \{\!|\, (x, r) \cdot x \in set(S) \cup set(T) \mid r = max(\#(x, S), \#(x, T)) \,|\!\} \quad \text{(Union)}$$
$$S \cap T \rightarrow \{\!|\, (x, min(a, b)) \cdot (x, a) \in S \wedge (x, b) \in T \mid a, b \geq 0 \,|\!\} \quad \text{(Intersection)}$$
$$S + T \rightarrow \{\!|\, (x, r) \cdot x \in set(S) \cup set(T) \mid r = \#(x, S) + \#(x, T) \,|\!\} \quad \text{(Sum)}$$
$$S - T \rightarrow \{\!|\, (x, r) \cdot (x, a) \in S \mid r = a - \#(x, T) \wedge r > 0 \,|\!\} \quad \text{(Difference)}$$
$$size(S) \rightarrow \sum (x, r) \in S \cdot r \quad \text{(Size)}$$

Additional notes and extended context:

The # Operator Defined as the number of occurrences of an element in a bag. If bags are represented by a relation, this corresponds to a direct lookup $\#(x, S) = S[x]$.

Intersection, Difference Since the intersection and difference operators are always decreasing (ex. $S \cap T \subseteq S \wedge S \cap T \subseteq T$ and $S - T \subseteq S$), we can short-circuit operations that would require looping over both sets instead of just S. *But how do we define this short-circuiting behaviour?* Intersections can make use of this property for both operands, but difference will always iterate over the first operand.

Difference $a - b > 0 \implies a > 0$.

Sum, Union The cast to set of $set(S)$ can be implemented by taking the domain of the bag-representing relations.

6 Implementation Representation

Different implementations of each data type will have varying strengths and weaknesses, not only in theoretical asymptotic time and space, but in concrete real-world tests. Cache usage and additional information through object metadata may prove influential on smaller tests. Since this document is only concerned with the theoretical compiler specification, we analyze the theoretical time and space complexity, then pair gathered examples with a test plan for hardware considerations.

A first approach to tackling these type representations would likely constitute a linked list. The space requirements for enumeration are straightforward, with extra allocations for link pointers. Insertions for unordered collections or append/concat operations are $O(1)$, but $O(n)$ for indexed insertion and union with one element. Lookups for all collections are $O(n)$, but this running time is undesirable for the often-used `in` operator for set-generated code. Since linked lists naturally enforce element order, this structure may be suitable for fast-changing sequences. Although, a limited-size sequence may be better suited for a contiguous array for $O(1)$ indexing. *TODO: For sequences, we should also see if trees/heaps or bloom filters could provide efficient membership checking. Bloom filters are probabilistic but can determine $\neq$ operations.*

On the other hand, hashmaps with $O(1)$ membership and element lookups are useful for all unordered collections. Relations may need bidirectional hashmaps that can efficiently handle many-to-many relations.

Compressed bitmaps may be used for sets, but require a lot of space for sparse elements.

Bags may be implemented either as a (linked) list, a set of tuples where the number of element occurrences is stored in the second tuple component, or a relation where the number of occurrences is the codomain.