Complete TRS Specification for Abstract Collection Types

Anthony Hunt

July 9, 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 $\{ x \cdot x \in S \land x \in T \}$, 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.

Granular Strategy (Sets)

Phase 1: Set Comprehension Construction Break down all qualifying sets into comprehension forms, collapsing and simplifying where needed.

Phase 2: 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

Phase 3: Predicate Simplification Remove superfluous dummy variables, group or-clauses that use the exact same generator (ex. $\{ x \cdot x \in S \land x \neq 0 \lor x \in S \land x = 0 \} \rightarrow \{ x \cdot x \in S \land (x \neq 0 \lor x = 0) \}$). Clauses should be group-able based on DNF, and generators should be selected and recorded.

Phase 4: 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	$S \nrightarrow T$	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	$R \Leftarrow Q$	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	$p \triangleleft S = \{ x \cdot x \in S \mid p(x) \}$
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	$S \cup T = \{\!\{ (x, a+b) \cdot (x, a) \in S \wedge (x, b) \in T \mid a, b \geq 0 \}\!\}$
Intersection	$S \cap T = \{\!\{ (x, min(a, b)) \cdot (x, a) \in S \wedge (x, b) \in T \mid a, b \geq 0 \}\!\}$
Difference	$S - T = \{\!\{ (x, a-b) \cdot (x, a) \in S \wedge (x, b) \in T \mid a, b \geq 0 \wedge a - b > 0 \}\!\}$
Bag Mapping	$f * S = \{\!\{ (f(x), r) \cdot (x, r) \in S \}\!\}$
Bag Filter	$p \triangleleft S = \{\!\{ (x, r) \cdot (x, r) \in S \mid p(x) \}\!\}$
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 $x \texttt{ in } S$ 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	$p \triangleleft S = [\, f(x) \cdot x \texttt{ in } S \mid p(x) \,]$
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	$R \lhd\!\!\lhd Q = Q \cup (dom(Q) \lhd R)$
(Forward) Composition	$Q \circ R = \{\, x \mapsto z \cdot x \mapsto y \in R \land y \mapsto z \in Q \,\}$
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	$S \lhd\!\!\!\cdot R = \{\, x \mapsto y \cdot x \mapsto y \in R \mid x \notin S \,\}$
Range Restriction	$R \rhd S = \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \in S \,\}$
Range Subtraction	$R \rhd\!\!\!\cdot S = \{\, x \mapsto y \cdot x \mapsto y \in R \mid y \notin S \,\}$

4 Rules

Below is a list of rewrite rules for key abstract data types and some builtin functions.

4.1 Builtin Functions

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

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

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

4.2 Sets

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

4.2.1 Phase 1: Set Comprehension Construction

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

[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.

4.2.2 Phase 2: Disjunctive Normal Form

$$a \wedge ... \wedge (b \wedge c) \wedge ... \rightsquigarrow a \wedge ... \wedge b \wedge c \wedge ... \quad \text{(Flatten Nested Ands)}$$
$$a \vee ... \vee (b \vee c) \vee ... \rightsquigarrow a \vee ... \vee b \vee c \vee ... \quad \text{(Flatten Nested Ors)}$$
$$\neg\neg x \rightsquigarrow x \quad \text{(Double Negation)}$$
$$\neg(x \vee y) \rightsquigarrow \neg x \wedge \neg y \quad \text{(Distribute De Morgan - Or)}$$
$$\neg(x \wedge y) \rightsquigarrow \neg x \vee \neg y \quad \text{(Distribute De Morgan - And)}$$
$$x \wedge (y \vee z) \rightsquigarrow (x \wedge y) \vee (x \wedge z) \quad \text{(Distribute Ands)}$$

4.2.3 Phase 3.1: Predicate Simplification 1

$$\{\, x, y \cdot P \wedge Q \mid E \,\} \rightsquigarrow \{\, x \cdot P \mid \{\, y \cdot Q \mid E \,\} \,\} \quad \text{(Nesting [a])}$$
$$\{\, x \cdot \bigwedge P_i \mid E \,\} \rightsquigarrow \{\, x \cdot \bigvee \bigwedge P_i \mid E \,\} \quad \text{(Or-wrapping [b])} \tag{1}$$

[a] y cannot occur in P

[b] 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.

4.2.4 Phase 3.2: Generator Selection 2

$$\bigwedge P_i \rightsquigarrow P_g \wedge \bigwedge P_a \wedge \bigwedge_{P_i \neq P_g, P_i \notin \bigcup P_a} P_i$$

(Generator Selection and Dummy Reassignment [a])

$$(P_g \wedge \bigwedge P_a \wedge \bigwedge P_i) \vee (P_g \wedge \bigwedge Q_a \wedge \bigwedge_{Q_i \neq P_g} Q_i) \rightsquigarrow P_g, \bigwedge P_a \wedge \bigwedge Q_a, \bigwedge P_i \vee \bigwedge Q_i$$

(Simplified DNF Form [b])

[a] The LH term must occur inside a quantifier's predicate - one match per or-clause. P_g is the generator, a single clause distinguished from the rest of $\bigwedge P_i$.
Dummy Reassignment uses assignment to calculate expressions outside of the if-statement. For example, $\{ x \cdot x \in S \wedge z = f(x) \mid z \}$ indirectly binds z (if z is free), although z does not appear in the quantifier list. This may be less efficient for simple cases than directly rewriting all occurrences of z to f(x), but additional conditions that make use of z would benefit from the intermediate calculation. P_a is an ordered list of such assignments (drawn from $\bigwedge P_i$), so that additional layers of indirection may be accommodated. The expectation is that LLVM will filter out superfluous assignments, though we will test this assumption later.
Right now, we just make a naive selection of generator (ie., the first viable option). Later, this will be more intelligent.

[b] The LH term must occur inside a quantifier's predicate. Combines clauses with the same generator. Requires Generator Selection to be run first

4.2.5 Phase 4: Set Code Generation

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

(Quantifier Generation [a])

[a] $\oplus$ works for any quantifier. The identity and accumulate functions are determined by the realized $\oplus$. For example, if $\oplus = \sum$, the identity is 0 and accumulate is addition.

$$\textbf{if } \bigvee P_i \textbf{ then} \\ \quad \text{body} \quad \rightsquigarrow \quad \begin{array}{l} \textbf{if } P_0 \textbf{ then} \\ \quad \text{body} \\ \textbf{if } P_1 \wedge \neg P_0 \textbf{ then} \\ \quad \text{body} \\ \ldots \end{array}$$

(Disjunct conditional)

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

(Conjunct conditional [a])

[a] Function *free* returns clauses in P that contain only free variables. *generator* is a single clause representing the selected generator of P (of form $x \in S$ where x will be bound by this loop condition). *bound* returns the clauses that contain the bound variable x.

$$a := accumulator(a, \oplus(E \mid P)) \quad \rightsquigarrow \quad \begin{array}{l}\textbf{if } P \textbf{ then} \\ \quad a := accumulator(a, E)\end{array}$$
(Accumulating Quantifier)

4.3 Relations

$$R[S] \to \{\, x \mapsto y \in R \cdot x \in S \mid y \,\} \quad \text{(Image)}$$
$$x \mapsto y \in S \times T \to x \in S \land y \in T \quad \text{(Product)}$$
$$x \mapsto y \in R^{-1} \to y \mapsto x \in R \quad \text{(Inverse)}$$
$$x \mapsto y \in (Q \circ R) \to x \mapsto z \in Q \land z' \mapsto y \in R \land z = z' \quad \text{(Composition)}$$
$$R \lhd\!\!\!+ Q \to Q \cup (dom(Q) \lhd R) \quad \text{(Override)}$$

$$dom(R) \to map(fst, R) \quad \text{(Domain)}$$
$$ran(R) \to map(snd, R) \quad \text{(Range)}$$

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

4.4 Bags

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

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

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. $S \cap T \subseteq S \land S \cap T \subseteq T$ 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.

5 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.