Verification of High-Level Transformations with Inductive Refinement Types

Anonymous Author(s)

Abstract

High-level transformation languages like Rascal include expressive features for manipulating large abstract syntax trees: first-class traversals, expressive pattern matching, backtracking and generalized iterators. We present the design and implementation of an abstract interpretation tool, Rabit, for verifying inductive type and shape properties for transformations written in such languages. We describe how to perform abstract interpretation based on operational semantics, specifically focusing on the challenges arising when analyzing the expressive traversals and pattern matching. Finally, we evaluate Rabit on a series of transformations (normalization, desugaring, refactoring, code generators, type inference, etc.) showing that we can effectively verify stated properties.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords transformation languages, abstract interpretation, static analysis

ACM Reference Format:
Anonymous Author(s). 2018. Verification of High-Level Transformations with Inductive Refinement Types. In *Proceedings of 17th International Conference on Generative Programming: Concepts & Experience (GPCE 2018).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.475/123_4

1 Introduction

Transformations play a central role in *software language engineering* [30]. They are used, amongst others, for desugaring, model transformations, refactoring, and code generation. The artifacts involved in transformations—e.g., structured data, domain-specific models, and code—often have large abstract syntax, spanning hundreds of syntactic elements, and a correspondingly rich semantics. Thus, writing transformations is a tedious and error prone process. Specialized languages and frameworks with high-level features have been developed to address this challenge of writing and maintaining transformations. These languages include Rascal [31], Stratego/XT [11], TXL [16], Uniplate [33] for Haskell and Kiama [48] for Scala. For example, Rascal combines a functional core language supporting state and exceptions, with constructs for processing of large structures.

```
1 public Script flattenBlocks(Script s) {
2     solve(s) {
3         s = bottom-up visit(s) {
4             case stmtList: [*xs,block(ys),*zs] =>
5                 xs + ys + zs
6         }
7     }
8     return s;
9 }
```

Figure 1. Transformation in Rascal that flattens all nested blocks in a statement

Figure 1 shows an example Rascal transformation program in Fig. 1 taken from a PHP analyzer[1]. This transformation program recursively flattens all blocks in a list of statements. The program uses the following core Rascal features:

- A *visitor* to traverse and rewrite all elements matching the stated pattern—statement lists containing a block—to a flattened list of statements. Visitors support various strategies, like bottom-up in the example which performs the traversal starting from leaf elements and ending with the root of the abstract tree.
- The *rich pattern matching* language to non-deterministically find blocks elements inside a list of statements, where the starred variable patterns $\star xs$ and $\star zs$ match arbitrary number of elements in the list, respectively before and after the block element. Rascal also allows non-linear matching, negative matching and specifying patterns that match deeply nested values.
- The solve-loop performing the rewrite until a fixed point is reached, e.g. the value of s does not change.

To rule out errors in transformations, we propose a static analysis for enforcing type and shape properties, so that target transformations produce output adhering to particular shape constraints. For example:

- The transformation preserves the constructors used in the input, and does not add or remove new types of PHP statements.
- The transformation produces flat statement lists, i.e., do not recursively contain any block.

To ensure such properties, a verification technique must reason about shapes of inductive data—also inside collections such as sets and maps—while still maintaining soundness

[1] https://github.com/cwi-swat/php-analysis

GPCE 2018, November 2018, Boston, Massachusetts, USA
2018. ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

and precision. It must also track other important aspects, like cardinality of collections, which interact with target language operations including pattern matching and iteration.

In this paper, we address the problem of verifying type and shape properties for high-level transformations written in Rascal and similar languages. We show how to design and implement a static analysis based on abstract interpretation. Concretely, our contributions are:

1. An abstract interpretation-based static analyzer—Rascal ABstract Interpretation Tool (Rabit)—that supports inferring types and inductive shapes for a large subset of Rascal.
2. An evaluation of Rabit on several program transformations: refactoring, desugaring, normalization algorithm, code generator, and language implementation of an expression language.
3. A modular design for abstract shape domains, that allows extending and replacing abstractions for concrete element types, e.g. extending the abstraction for lists to include length in addition to shape of contents.
4. Formal Schmidt-style abstract *operational* semantics [45] for a significant subset of Rascal adapting the idea of *trace memoization* to support arbitrary recursive calls with input from infinite domains.

Together, these contributions show feasibility of applying abstract interpretation for constructing analyses for expressive transformation languages and properties.

We proceed by presenting a motivating example in Sect. 2. We formally introduce key constructs of Rascal in Sect. 3. Section 4 describes the modular construction of abstract domains. Sections 5 to 8 describe abstract semantics for the key constructs. We evaluate the analyzer on realistic transformations, reporting results in Sect. 9. Sections 10 and 11 discuss related papers and conclude.

2 Motivation and Overview

Verifying types and state properties such as the ones stated for the flattening program (Fig. 1), poses the following key challenges:

- The programs use *heterogeneous inductive data types*, and contain *collections* such as lists, maps and sets, and basic data such as integers and strings. This complicates construction of the abstract domains, since it must model interaction between these different types while maintaining precision.
- Execution of syntax tree traversals is heavily dependent on the *type and shape of input*, a *complex program state*, and involves *unbounded recursion*. This further constraints possible analyses in inferring approximate invariants using a procedure that is both terminating and provides useful results.
- Backtracking and exceptions in large programs introduce the possibility of *state-dependent non-local jumps* during execution, which makes it difficult to statically calculate the control flow of target programs.

```
1 data Nat = zero() | suc(Nat pred);
2 data Expr = var(str nm) | cst(Nat vl)
3           | mult(Expr el, Expr er);
4
5 Expr simplify(Expr expr) =
6   bottom-up visit (expr) {
7       case mult(cst(zero()), y) => cst(zero())
8       case mult(x, cst(zero())) => cst(zero())
9   };
```

Figure 2. The running example: eliminating multiplications by zero from expressions

Figure 2 presents a small pedagogical example using visitors. The example is an expression simplification transformation program, which traverses an expression tree bottom-up and reduces multiplications involving the constant 0 to 0. We now give an overview of the key analysis techniques contributed in this paper, explaining them using the simplification transformation as running example.

Inductive refinement types Rabit works by inferring an inductive refinement type representing the shape of possible output of a transformation given the shape of its input. It does this by interpreting the simplification program abstractly, considering all possible paths the program can take for values satisfying the input shape (any expression of type Expr in this case). The result of running Rabit on this case is:

$$\mathsf{success}\ \mathsf{cst}\,(\mathsf{Nat}) \wr \mathsf{var}\,(\mathsf{str}) \wr \mathsf{mult}\,(\mathsf{Expr}', \mathsf{Expr}')$$
$$\mathsf{fail}\ \mathsf{cst}\,(\mathsf{Nat}) \wr \mathsf{var}\,(\mathsf{str}) \wr \mathsf{mult}\,(\mathsf{Expr}', \mathsf{Expr}')$$

where $\mathsf{Expr}' = \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{Nat})) \wr \mathsf{var}\,(\mathsf{str}) \wr \mathsf{mult}\,(\mathsf{Expr}', \mathsf{Expr}')$ If the input was rewritten during traversal (success) then the resulting expression tree contains no multiplications by zero; if the traversal failed to match anywhere (fail), then the input value did not contain any multiplication by zero to begin with. This can be observed by seeing that the last alternative $\mathsf{mult}\,(\mathsf{Expr}', \mathsf{Expr}')$ contains only expressions of inductive refinement type Expr', which only allows multiplications by constants which are constructed using $\mathsf{suc}\,(\mathsf{Nat})$ (that is ≥ 1). We now proceed discussing how Rabit can infer this shape using abstract interpretation.

Abstractly interpreting traversals The core idea of abstractly executing a traversal is similar to concrete execution: we recursively traverse the input structure and rewrite the values that match target patterns. However, because of abstraction we must make sure to take into account all applicable paths. Figure 3 shows the execution tree of the traversal on the simplification example (Fig. 2) with shape $\mathsf{mult}\,(\mathsf{cst}\,(\mathsf{Nat})\,, \mathsf{cst}\,(\mathsf{Nat}))$. Since there is only one constructor, it will initially *recurse* down to traverse the contained

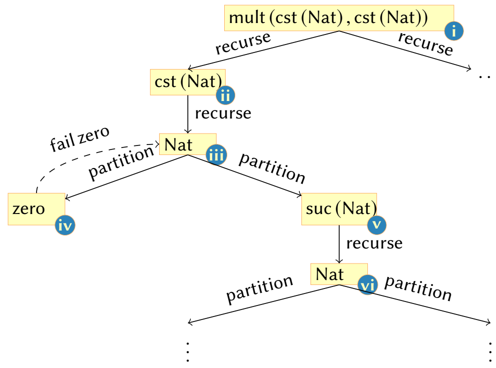

Figure 3. Naively abstractly interpreting the simplification example from Fig. 2 with initial input mult (cst (Nat) , cst (Nat)). The procedure does not terminate because of infinite recursion on Nat.

values (children) creating a new recursion node in the figure (ii) containing the left child cst (Nat), and then recurse again to create a node (iii) containing Nat. Observe here that Nat is an abstract type with two possible constructors (zero, suc (·)), and it is unknown at time of abstract interpretation, which of these constructors we have. When Rabit hits a type or a choice between alternative constructors, it explores each alternative separately creating new *partition* nodes; in our example we partition the Nat type into its constructors zero (node iv) and suc (Nat) (node v). The zero case now represents the first case without children and we can run the visitor operations on it; since no pattern matches zero it will return a fail zero result indicating that it has not been rewritten. For the suc (Nat) case it will try to recurse down to Nat (node vi) which is equal to (node iii); if we continue our traversal algorithm as is, we will not terminate and get a result. To provide a terminating algorithm we will resort to using *trace memoization*.

Partition-driven trace memoization The core idea of trace memoization [43, 45] is to detect the paths where execution recursively meets similar input, merging the new recursive node with the similar previous one, thus creating a loop in the execution tree. This loop is then resolved by a fixed-point iteration.

In Rabit, we propose *partition-drive trace memoization*, which works with potentially unbounded input like the inductive type refinements that are supported by our abstraction. The technique detects cycles by keeping a *memoization map* which for each type—used for partitioning—stores the last traversed value (input) and the last result produced for this value (output). This memoization map is initialized to map all types to the bottom element (⊥) for both input and output. The evaluation is modified to use the memoization map, so it checks on each iteration the input i against the memoization map:

- If it finds that the last processed input i' for the same type is greater than the current input ($i' \sqsupseteq i$), then it uses the corresponding output; i.e., we found a hit in the memoization map.
- Otherwise, it will merge the last processed and current inputs to a new value $i'' = i' \nabla i$, update the memoization map and continue execution with i''. The merging operation (∇) is called a widening; it ensures that the result is an upper bound of its inputs, i.e., $i' \sqsubseteq i'' \sqsupseteq i$ and that the merging will eventually terminate for the increasing chain of values.

We demonstrate the trace memoization and fixed-point iteration procedures on Nat in Fig. 4, beginning with the leftmost tree. The expected result is fail Nat, meaning that no pattern has matched, no rewrite has happened, and a value of type Nat is returned, since the simplification program only introduces changes to values of type Expr.
We represent the memoization map inside a framed box. The result of the widening is presented below the memoization map. In all cases the widening in Fig. 4 is trivial, as it happens against ⊥. The final line in node 1 stores the value o_{prev} produced by the previous iteration of the traversal, to establish whether a fixed point has been reached (also ⊥ initially).

Trace partitioning As before, we *partition* [41] the abstract value Nat along its constructors: zero and suc (·). This partitioning is important to maintain precision through the abstract interpretation. As in Fig. 3, the left branch fails immediately, as no pattern in Fig. 2 matches zero. The right branch descends into a new recursion over Nat, but now with an updated memoization table (Fig. 4). This run terminates, due to a hit in the memoization map, returning ⊥. After returning, the value of suc (Nat) should be reconstructed with the result of the rewrite. Since the result is ⊥, it is just propagated upwards. At the return to the last widening node, the values are joined, and widen the previous iteration result o_{prev} (the dotted arrow on top). This process repeats in the second and third iterations, but now the reconstruction in node 3 succeeds. In the third iteration, we join and widen the following components (cf., o_{prev} and the dashed arrows incoming into node 1 in the rightmost column):

$$[\text{zero} \wr \text{suc}\,(\text{zero})\ \nabla\ (\text{zero} \sqcup \text{suc}\,(\text{zero} \wr \text{suc}\,(\text{zero})))] = \text{Nat}$$

Here, the used widening operator [18] accelerates the convergence by increasing the value to represent the entire type Nat. It is easy to convince yourself, by following the same recursion steps as in the figure, that the next iteration, using o_{prev} = Nat will produce Nat again, arriving at a fixed point. Observe, how consulting the memoization map, and widening the current value accordingly, allowed us to avoid infinite recursion over unfoldings of Nat.

Nesting fixed point iterations. In cases, where inductive shapes (e.g., Expr) refer to other inductive shapes (e.g., Nat),

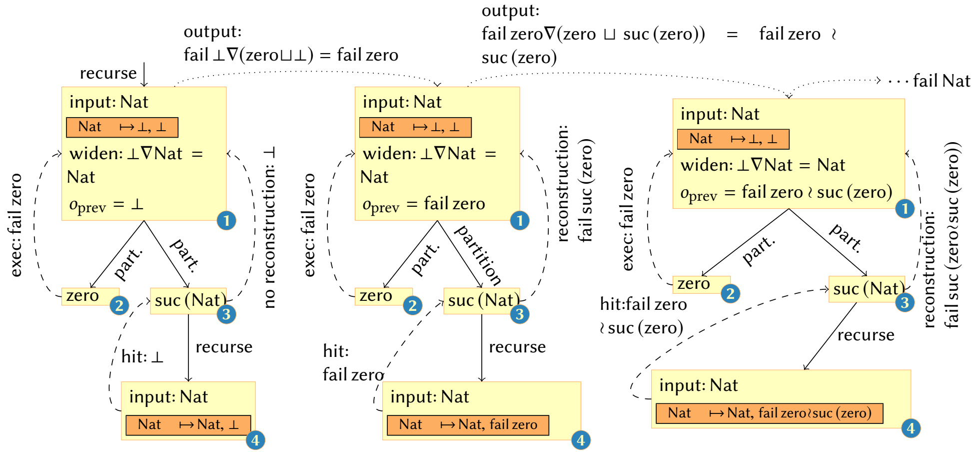

Figure 4. Three iterations of a fixed point computation for input Nat. Iterations are separated by dotted arrows on top

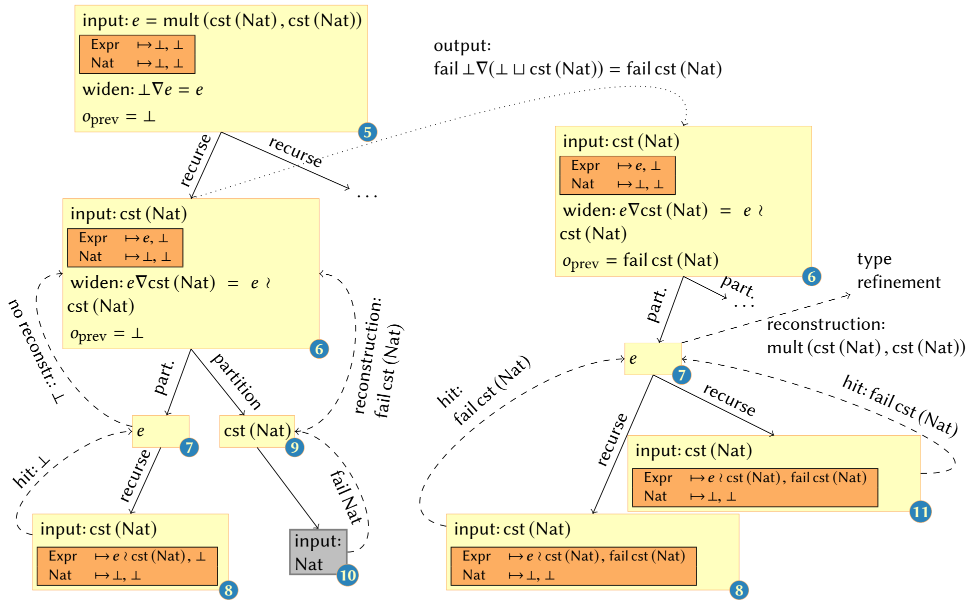

Figure 5. A prefix of the abstract interpreter run for e = mult (cst (Nat) , cst (Nat)). Fragments of two iterations involving node 6 are shown, separated by a dotted arrow.

it is usually necessary to perform nested fixed-point iterations to solve recursion at each level. Figure 5 returns to the more high-level fragment of the traversal of Expr, starting with mult (cst (Nat) , cst (Nat)) (as in Fig. 3). We follow the recursion tree along nodes 5, 6, 7, 8, 9, 10, 9, 6, following the same rules as in Fig. 4. In node 10 we run a nested fixed point iteration on Nat, already discussed in Fig. 4, so we just include the final result.

Type refinement. The output of the first iteration in node 6 is fail cst (Nat), which becomes the new o_{prev}, and the second iteration begins (to the right). After the widening the input is partitioned into e (node 7) and cst (Nat)(node elided). When the second iteration returns to node 7 we have the following reconstructed value: mult (cst (Nat) , cst (Nat)). Contrast, this with lines 6-7 in Fig. 2, to see that running the abstract value against this pattern might actually produce success. In order to obtain precise result shapes, we refine the input values when they fail to match a pattern. Our abstract interpreter produces a refinement of the type, by running it through the pattern matching, giving:

$$\text{success cst (Nat)}$$
$$\text{fail mult (cst (suc (Nat)) , cst (suc (Nat)))}$$

The result means, that if the pattern match succeeds then it produces an expression of type cst (Nat). More interestingly,

if the matching failed neither the left nor the right argument of mult $(\cdot,\cdot)$ could have contained the constant zero—the interpreter captured some aspect of the semantics of the program by *refining* the input type. Naturally, from this point on the recursion and iteration continues, but we shall abandon the example, and move on to formal developments.

3 Formal Language

The presented technique is meant to be general and applicable to many high-level transformation languages. However, to keep the presentation concise, we focus on few key constructs from Rascal [31], relying on the concrete semantics from Rascal Light [2].

We consider algebraic data types (at) and finite sets ($\mathsf{set}\langle t\rangle$) of elements of type t. Each algebraic data type, at has a set of unique constructors. Each constructor $k(\underline{t})$ has a fixed set of typed parameters. The language includes sub-typing, with void and value as bottom and top types respectively.

$$t \in \mathsf{Type} ::= \mathsf{void} \mid \mathsf{set}\langle t\rangle \mid at \mid \mathsf{value}$$

We consider the following subset of Rascal expressions: From left to right we have: variable access, assignments, sequencing, constructor expressions, set literal expressions, matching failure expression, and bottom-up visitors:

$$e ::= x \in \mathsf{Var} \mid x = e \mid e;e \mid k(\underline{e}) \mid \{\underline{e}\} \mid \mathsf{fail} \mid \mathsf{bu\ visit}\ e\ \underline{cs}$$
$$cs ::= \mathsf{case}\ p \Rightarrow e$$

Visitors are a key construct in Rascal. A visitor bu visit e $\underline{cs}$ recursively traverses an input value obtained by evaluating e (any combination of simple values, data type values and collections). During the traversal, case expression $\underline{cs}$ are applied to the nodes, and the values matching target patterns are rewritten. We will discuss a concrete subset of patterns p further in Sect. 6. For brevity, we only discuss the bottom-up visitors in the paper. However, Rabit (Sect. 9) supports all strategies support by Rascal.

Notation We write $(x,y) \in f$ to denote the pair (x,y) where $x \in \mathrm{dom}\, f$ and $y = f(x)$. Abstract semantic components, sets and operations are marked with a hat $\widehat{a}$. A sequence of $e_1,\ldots,e_n$ is represented by an underline $\underline{e}$. The empty sequence is denoted by ε, and concatenation of sequences $\underline{e_1}$ and $\underline{e_2}$ is written $\underline{e_1}, \underline{e_2}$. Notation is lifted to sequences in an intuitive manner: for example given a sequence $\underline{v}$, the value v_i denotes the ith element in the sequence, and $\underline{v:t}$ denotes the sequence $v_1:t_1,\ldots,v_n:t_n$.

4 Abstract Domains

Our abstract domains are designed to allow modular composition. Modularity is key for transformation languages, which manipulate a large variety of kinds of values. The design allows easily replacing abstract domains for particular types of values, as well as adding support for new value types. Our goal is to construct an abstract value domain $\widehat{vs} \in \widehat{\mathsf{ValueShape}}$ which captures inductive refinement types of form:

$$at^r = k_1(\underline{\widehat{vs}_1}) \wr \cdots \wr k_n(\underline{\widehat{vs}_n})$$

where each value $\widehat{vs}_i$ can possibly recursively refer to at^r. Below, we define abstract domains for sets, data types and recursively defined domains.

The modular domain design generalizes parameterized domains [17] to follow a design inspired by the modular construction of types and domains [7, 15, 46]. The idea is to define domains parametrically—i.e. in the form $\widehat{\mathsf{F}}(\underline{\widehat{\mathsf{E}}})$—so that abstract domains for subcomponents are taken as parameters, and explicit recursion is handled separately. We use standard domain combinators [53] to combine the various domains into our target abstract value domain.

Set shape domain. We abstract finite sets using abstract elements $\{\widehat{e}\}_{[l;u]}$ from a parameterized domain $\widehat{\mathsf{SetShape}}(\widehat{\mathsf{E}})$. The component from the parameter domain ($\widehat{e} \in \widehat{\mathsf{E}}$) represents the abstraction of the shape of elements, and a non-negative interval component $[l;u] \in \widehat{\mathsf{Interval}}^{+}$ is used to abstract over the cardinality. The abstract set element acts as a reduced product between $\widehat{e}$ and $[l;u]$ and thus the lattice operations follow directly from each component.

Given a concretization function for the abstract content domain $\gamma_{\widehat{\mathsf{E}}} \in \widehat{\mathsf{E}} \rightarrow \wp(\mathsf{E})$, we can define a concretization function for the abstract set shape domain to possible finite sets of concrete elements $\gamma_{\widehat{\mathsf{SS}}} \in \widehat{\mathsf{SetShape}}(\widehat{\mathsf{E}}) \rightarrow \wp(\mathsf{Set}\,(\mathsf{E}))$:

$$\gamma_{\widehat{\mathsf{SS}}}(\{\widehat{e}\}_{[l;u]}) = \left\{ es \;\middle|\; es \subseteq \gamma_{\widehat{\mathsf{E}}}(\widehat{e}) \wedge |es| \in \gamma_{\widehat{\mathsf{I}}}([l;u]) \right\}$$

Example 4.1. We can concretize abstract elements from $\widehat{\mathsf{SetShape}}(\widehat{\mathsf{Interval}})$ to a set of possible sets of integers $\wp(\mathsf{Set}\,(\mathbb{Z}))$ as follows: $\gamma_{\widehat{\mathsf{SS}}}(\{[42;43]\}_{[1;2]}) = \{\{42\},\{43\},\{42,43\}\}$

Data shape domain. Inductive refinement types are defined as a generalization of refinement types [23, 44, 55] that inductively constrain the possible constructors and the content in a data structure. We use a parameterized abstraction of data types $\widehat{\mathsf{DataShape}}(\widehat{\mathsf{E}})$, whose parameter $\widehat{\mathsf{E}}$ abstracts over the shape of constructor arguments:

$$\widehat{d} \in \widehat{\mathsf{DataShape}}(\widehat{\mathsf{E}}) = \{\bot_{\widehat{\mathsf{DS}}}\} \cup \{k_1(\underline{e_1}) \wr \ldots \wr k_n(\underline{e_n}) \mid e_i \in \widehat{\mathsf{E}}\} \cup \{\top_{\widehat{\mathsf{DS}}}\}$$

We have the least element $\bot_{\widehat{\mathsf{DS}}}$ and top element $\top_{\widehat{\mathsf{DS}}}$ elements—respectively representing no data types value and all data type values—and otherwise a non-empty choice between unique (all different) constructors of the same algebraic data type $k_1(\underline{e_1}) \wr \cdots \wr k_n(\underline{e_n})$ (shortened $\underline{k(\underline{e})}$). We can treat the constructor choice as a finite map $[k_1 \mapsto \underline{e_1},\ldots,k_n \mapsto \underline{e_n}]$, and then directly define our lattice operations point-wise.

Given a concretization function for the concrete content domain $\gamma_{\widehat{\mathsf{E}}} \in \widehat{\mathsf{E}} \rightarrow \wp(\mathsf{E})$, we can create a concretization function for the data shape domain $\gamma_{\widehat{\mathsf{DS}}} \in \widehat{\mathsf{DataShape}}(\widehat{\mathsf{E}}) \rightarrow \wp(\mathsf{Data}(\mathsf{E}))$ where $\mathsf{Data}(\mathsf{E}) = \left\{ k(\underline{v}) \;\middle|\; \exists at.k(\underline{v}) \in [\![at]\!] \wedge \underline{v \in \mathsf{E}} \right\}$

The concretization is defined as follows:

$$\gamma_{\widehat{\mathsf{DS}}}(\bot_{\widehat{\mathsf{DS}}}) = \emptyset \qquad \gamma_{\widehat{\mathsf{DS}}}(\top_{\widehat{\mathsf{DS}}}) = \mathsf{Data}(\mathsf{E})$$
$$\gamma_{\widehat{\mathsf{DS}}}(k_1(\underline{e_1}) \wr \cdots \wr k_n(\underline{e_n})) = \left\{ k_i(\underline{v}) \;\middle|\; i \in [1, n] \wedge \underline{v \in \gamma_{\widehat{\mathsf{E}}}(e_i)} \right\}$$

Example 4.2. Given $\widehat{\mathsf{Interval}}$ as the base domain, we can concretize abstract data elements $\widehat{\mathsf{DataShape}}(\widehat{\mathsf{Interval}})$ to a set of possible concrete data values $\wp(\mathsf{Data}(\mathbb{Z}))$. Consider values from the algebraic data type:

$$\mathsf{data\ errorloc} = \mathsf{repl}() \mid \mathsf{linecol}(\mathsf{int}, \mathsf{int})$$

We can concretize abstracting elements as follows:

$$\gamma_{\widehat{\mathsf{DS}}}(\mathsf{repl}() \wr \mathsf{linecol}(1, [3; 4])) = \{\mathsf{repl}(), \mathsf{linecol}(1, 3), \mathsf{linecol}(1, 4)\}$$

Recursive shapes. We extend our abstract domains to cover recursive structures such as lists and trees. Given a type expression $\mathsf{F}(X)$ with a variable X, we construct the abstract domain as the solution to the recursive equation $X = \mathsf{F}(X)$ [46, 49, 53], obtained by iterating the induced map F over the empty domain $\mathbb{0}$ and adjoining a new top element to the limit domain. The concretization function of the recursive domain follows directly from the concretization function of the underlying functor domain.

Example 4.3. We can concretize abstract elements of the refinement type from our running example:

$$\gamma_{\widehat{\mathsf{DS}}}(\mathsf{Expr}^e) = \left\{ \begin{array}{c} \overbrace{\mathsf{cst}(\mathsf{suc}(\mathsf{suc}(\mathsf{zero})))}^{2}, \mathsf{mult}(2, 2), \\ \mathsf{mult}(\mathsf{mult}(2, 2), 2), \ldots \end{array} \right\}$$

where $\mathsf{Expr}^e = \mathsf{cst}(\mathsf{suc}(\mathsf{suc}(\mathsf{zero}))) \wr \mathsf{mult}(\mathsf{Expr}^e, \mathsf{Expr}^e)$ In particular, our abstract element represents the set of all multiplications of the constant 2.

Value domains. We presented the required components for abstracting individual types, and now all that is left is putting everything together. We construct our value shape domain using choice and recursive domain equations:

$$\widehat{\mathsf{ValueShape}} = \widehat{\mathsf{SetShape}}(\widehat{\mathsf{ValueShape}}) \oplus \widehat{\mathsf{DataShape}}(\widehat{\mathsf{ValueShape}})$$

Similarly, we have the corresponding concrete shape domain:

$$\mathsf{Value} = \mathsf{Set}(\mathsf{Value}) \uplus \mathsf{Data}(\mathsf{Value})$$

We then have a concretization function $\gamma_{\widehat{\mathsf{VS}}} \in \widehat{\mathsf{ValueShape}} \to \wp(\mathsf{Value})$, which follows directly from the previously defined concretization functions.

Abstract state domains

We now explain how do we construct abstractions states and results for executing Rascal programs abstractly.

Abstract store domain. Tracking assignments of variables is important since matching variable patterns depends on the value being assigned in the store:

$$\widehat{\sigma} \in \widehat{\mathsf{Store}} = \mathsf{Var} \to \{\mathsf{ff}, \mathsf{tt}\} \times \widehat{\mathsf{ValueShape}}$$

For a variable x we get $\widehat{\sigma}(x) = (b, \widehat{vs})$ where b is true if x might be unassigned, and false otherwise (when x is definitely assigned). The second component, $\widehat{vs}$ is a shape approximating a possible value of x.

We lift the orderings and lattice operations point-wise from the value shape domain to abstract stores. We define the concretization function $\gamma_{\widehat{\mathsf{Store}}} \in \widehat{\mathsf{Store}} \to \wp(\mathsf{Store})$ as follows:

$$\gamma_{\widehat{\mathsf{Store}}}(\widehat{\sigma}) = \left\{ \sigma \;\middle|\; \begin{array}{c} \forall x, b, \widehat{vs}. \widehat{\sigma}(x) = (b, \widehat{vs}) \Rightarrow \\ (\neg b \Rightarrow x \in \mathbf{dom}\ \sigma) \\ \wedge (x \in \mathbf{dom}\ \sigma \Rightarrow \sigma(x) \in \gamma_{\widehat{\mathsf{V}}}(\widehat{vs})) \end{array} \right\}$$

Abstract result domain. Traditionally, abstract control flow is often handled using a denotational collecting semantics with continuations or by explicitly constructing a control flow graph. These approaches are non-trivial to apply for a rich language like Rascal, which has backtracking, exceptions and data-dependent control flow introduced by visitors. A nice side-effect of Schmidt-style abstract interpretation is that it allows handling abstraction of control flow directly.

We model different type of results—successes, pattern match failures, errors directly in a $\widehat{\mathsf{ResultSet}}$ domain which keeps track of possible results with each its own separate store. Keeping separate stores is cheap since the number of result types is fixed, and is important to maintain precision around different paths:

$$rest \in \widehat{\mathsf{ResType}} ::= \mathsf{success} \mid exres$$
$$exres \in \mathsf{ExcType} ::= \mathsf{fail} \mid \mathsf{error} \qquad \widehat{resv} \in \widehat{\mathsf{ResVal}} ::= \cdot \mid \widehat{vs}$$
$$\widehat{Res} \in \widehat{\mathsf{ResultSet}} = \widehat{\mathsf{ResultType}} \rightharpoonup \widehat{\mathsf{ResultValue}} \times \widehat{\mathsf{Store}}$$

The lattice operations are lifted directly from the target value domains and store domains. We define the concretization function $\gamma_{\widehat{\mathsf{RS}}} \in \widehat{\mathsf{ResultSet}} \to \wp(\mathsf{Result} \times \mathsf{Store})$:

$$\gamma_{\widehat{\mathsf{RS}}}(\widehat{Res}) = \left\{ (rest\ resv, \sigma) \;\middle|\; \begin{array}{c} (rest, (\widehat{resv}, \widehat{\sigma})) \in \widehat{Res} \wedge \\ resv \in \gamma_{\widehat{\mathsf{RV}}}(\widehat{resv}) \wedge \sigma \in \gamma_{\widehat{\mathsf{Store}}}(\widehat{\sigma}) \end{array} \right\}$$

5 Abstract Semantics

A distinguishing feature of Schmidt-style abstract interpretation is that the derivation of abstract operational rules from a given concrete operational semantics is systematic and to a large extent mechanisable [9, 45]. The creative work is therefore reduced to providing abstract definitions for conditions and semantic operations such as pattern matching, and defining *trace memoization strategies* for non-structurally recursive operational rules, to finitely approximate an infinite

number of concrete traces and produce a terminating static analysis.

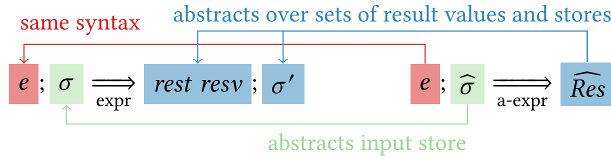

Figure 6. Relating concrete semantics (left) to abstract semantics (right).

Figure 6 relates the concrete evaluation judgment (left) to the abstract evaluation judgment (right) for Rascal expressions. Both judgements evaluate the same expression e. The abstract evaluation judgment abstracts the initial concrete store σ with an abstract store $\widehat{\sigma}$. The result of the abstract evaluation is a finite result set $\widehat{Res}$, abstracting over possibly infinitely many concrete result values *rest resv* and stores σ'. $\widehat{Res}$ maps each result type *rest* to a pair of abstract result value $\widehat{resv}$ and abstract result store $\widehat{\sigma}'$, i.e.:

$$\widehat{Res} = [rest_1 \mapsto (\widehat{resv}_1, \widehat{\sigma}_1), \ldots, rest_n \mapsto (\widehat{resv}_n, \widehat{\sigma}_n)]$$

There is an important difference in how the concrete and abstract semantic rules are used. In a concrete operational semantics a language construct is usually evaluated as soon as the premises of a rule are satisfied. When evaluating abstractly, we must consider *all* applicable rules, to soundly over-approximate the possible concrete executions. To this end, we introduce a special notation to collect all derivations with the same input i into a single derivation with output O equal to the join of the individual outputs:

$$\{\!|\, i \Rightarrow O \,|\!\} \triangleq O = \bigsqcup \{o \mid i \Rightarrow o\}$$

Let's use the operational rules for variable accesses to illustrate the steps in Schmidt-style translation of operational rules. The concrete semantics contains two rules for variable accesses, E-V-S for successful lookup, and E-V-Er for producing errors when accessing unassigned variables:

$$\text{E-V-S}\ \frac{x \in \operatorname{dom} \sigma}{x; \sigma \xRightarrow[\text{expr}]{} \mathsf{success}\ \sigma(x); \sigma} \qquad \text{E-V-Er}\ \frac{x \notin \operatorname{dom} \sigma}{x; \sigma \xRightarrow[\text{expr}]{} \mathsf{error}; \sigma}$$

We follow three steps, to translate the concrete rules to abstract operational rules:

1. For each concrete rule, create an abstract rule that uses a judgment for evaluation of a syntactic form, e.g., AE-V-S and AE-V-Er for variables.
2. Replace the concrete conditions and semantic operations with the equivalent abstract conditions and semantic operations for target abstract values, e.g. $x \in \operatorname{dom} \sigma$ with $\widehat{\sigma}(x) = (b, \widehat{vs})$ and a check on b. We obtain two execution rules:

$$\text{AE-V-S}\ \frac{\widehat{\sigma}(x) = (b, \widehat{vs})}{x; \widehat{\sigma} \xRightarrow[\text{a-expr-v}]{} [\mathsf{success} \mapsto (\widehat{vs}, \widehat{\sigma})]}$$

$$\text{AE-V-ER}\ \frac{\widehat{\sigma}(x) = (\mathsf{tt}, \widehat{vs})}{x; \widehat{\sigma} \xRightarrow[\text{a-expr-v}]{} [\mathsf{error} \mapsto (\cdot, \widehat{\sigma})]}$$

   Observe when b is true, both a success and failure may occur. So the abstract execution is non-deterministic.
3. Create a rule that collects all possible evaluations of the syntax-specific judgment rules, e.g. AE-V for variables:

$$\text{AE-V}\ \frac{\{\!|\, x; \widehat{\sigma} \xRightarrow[\text{a-expr-v}]{} \widehat{Res}' \,|\!\}}{x; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}'}$$

The possible shapes of the result value depend on the pair assigned to x in the abstract store. If the value shape of x is $\bot$, we drop the success result from the result set. The following examples illustrate the possible outcome result shapes:

Assigned Value	Result Set	Rules
$\widehat{\sigma}(x) = (\mathsf{ff}, \bot_{\widehat{VS}})$	$[]$	AE-V-S
$\widehat{\sigma}(x) = (\mathsf{ff}, [1;3])$	$[\mathsf{success} \mapsto ([1;3], \widehat{\sigma})]$	AE-V-S
$\widehat{\sigma}(x) = (\mathsf{tt}, \bot_{\widehat{VS}})$	$[\mathsf{error} \mapsto (\cdot, \widehat{\sigma})]$	AE-V-S, AE-V-Er
$\widehat{\sigma}(x) = (\mathsf{tt}, [1;3])$	$[\mathsf{success} \mapsto ([1;3], \widehat{\sigma}),$ $\mathsf{error} \mapsto (\cdot, \widehat{\sigma})]$	AE-V-S, AE-V-Er

It is possible to translate the operational semantics rules for other basic expressions using the presented steps (see ??). The core changes are the ones moving from checks of definiteness to checks of *possibility*. For example:

- Checking that evaluation of e has succeeded, requires that the abstract semantics uses $e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}$ and $(\mathsf{success}, (\widehat{vs}, \widehat{\sigma}')) \in \widehat{Res}$, as compared to $e; \sigma \xRightarrow[\text{expr}]{} \mathsf{success}\ v; \sigma'$ in the concrete semantics.
- Typing is now done using abstract judgments $\widehat{vs} \mathrel{\widehat{:}} t$ and $t \mathrel{\widehat{<:}} t'$. In particular, type t is an abstract subtype of type t' ($t \mathrel{\widehat{<:}} t'$) if there is a subtype t'' of t ($t'' <: t$) that is also a subtype of t' ($t'' <: t'$). This implies that $t \mathrel{\widehat{<:}} t'$ and $t \mathrel{\widehat{\not<:}} t'$ are non-exclusive.
- To check whether a particular constructor is possible, we use the abstract auxiliary function $\widehat{\mathrm{unfold}}(\widehat{vs}, t)$ which produces a refined value of type t if possible—splitting alternative constructors for data type values—and additionally produces error if the value is possibly not an element of t.

6 Pattern Matching

Expressive pattern matching is key feature of high-level transformation languages. Rabit handles the full Rascal pattern language including type-based matching and deep pattern matching. For brevity, we discuss a subset, including variables x, constructor patterns $k(\underline{p})$, and set patterns $\{\underline{\star p}\}$:

$$p ::= x \mid k(\underline{p}) \mid \{\underline{\star p}\} \qquad \star p ::= p \mid \star x$$

Rascal allows non-linear matching where the same variable x can be mentioned more than once: all values matched against x must have equal values for the match to succeed. Each set pattern contains a sequence of sub-patterns $\star p$; a sub-pattern is either an ordinary pattern p matched against a single set element, or a star pattern $\star x$ to be matched against a subset of elements. Star patterns can backtrack when pattern matching fails because of non-linear variable references, or when explicitly triggered by the fail expression.

This expressiveness poses challenges for developing an abstract interpreter that is not only sound, but is also sufficiently *precise* to prove interesting properties. We discuss the key aspects of Rabit how handles pattern matching, highlighting how we maintain precision by *refining* input values on pattern matching successes and failures.

Satisfiability semantics for patterns We begin by defining what it means that a (concrete/abstract) value matches a pattern. Figure 7a shows the concrete semantics for patterns. In the figure, ρ is a binding environment:

$$\rho \in \mathsf{BindingEnv} = \mathsf{Var} \rightharpoonup \mathsf{Value}$$

We say that a value v matches a pattern p ($v \models p$) if and only if there exists a binding environment ρ that maps the variables in the pattern to values in $\mathbf{dom}\,\rho = \mathsf{vars}(p)$ so that v is accepted by the satisfiability semantics $v \models_\rho p$ as defined in the figure. Constructor patterns $k(\underline{p})$ accept any well-typed value $k(\underline{v})$ of the same constructor whose subcomponents $\underline{v}$ match the sub-patterns $\underline{p}$ consistently in the same binding environment ρ. A variable x matches exactly the value it is bound to in the binding environment ρ. A set pattern $\{\underline{\star p}\}$ accepts any set of values $\{\underline{v}\}$ such that an associative-commutative arrangement of the sub-values $\underline{v}$ matches the sequence of sub-patterns $\underline{\star p}$ under ρ.

A value sequence $\underline{v}$ matches a pattern sequence $\underline{\star p}$ ($\underline{v} \models^\star \underline{\star p}$) if there exists a binding environment ρ such that $\mathbf{dom}\ \rho = \mathsf{vars}(\underline{\star p})$ and $\underline{v} \models^\star_\rho \underline{\star p}$. An empty sequence of patterns ε accepts an empty sequence of values ε. A sequence starting $p, \underline{\star p'}$ with an ordinary pattern p matches any non-empty sequence of values $v, \underline{v'}$ where v matches p and $\underline{v'}$ matches $\underline{\star p'}$ consistently under the same binding environment ρ. A sequence $\star x, \underline{\star p'}$ works analogously but it splits the value sequence in two $\underline{v}$ and $\underline{v'}$, such that x is assigned to $\underline{v}$ in ρ and $\underline{v'}$ matches $\underline{\star p'}$ consistently in ρ.

Example 6.1. Let's revisit the running example to understand how the data type values are matched. We consider matching the following (singleton) set of expression values:

$$\{\overbrace{\mathsf{mult}\,(\mathsf{cst}\,(\mathsf{zero})\,, \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero})))\,, \mathsf{cst}\,(\mathsf{zero})}^{\underline{v}}\}$$

against the pattern $p = \{\mathsf{mult}\,(x, y)\,, \star w, x\}$ in the environment $\rho = [x \mapsto \mathsf{cst}\,(\mathsf{zero})\,, y \mapsto \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero}))\,, w \mapsto \{\}]$. The matching argument is as follows:

$$\begin{aligned}&\{\underline{v}\} \models_\rho p \ \mathbf{iff}\ \underline{v} \models^\star_\rho \mathsf{mult}\,(x, y)\,, \star w, x\\ &\mathbf{iff}\ \mathsf{mult}\,(\mathsf{cst}\,(\mathsf{zero})\,, \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero}))) \models_\rho \mathsf{mult}\,(x, y)\\ &\mathbf{and}\ \mathsf{cst}\,(\mathsf{zero}) \models^\star_\rho \star w, x\end{aligned}$$

We see that the first conjunct matches as follows:

$$\begin{aligned}&\mathsf{mult}\,(\mathsf{cst}\,(\mathsf{zero})\,, \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero}))) \models_\rho \mathsf{mult}\,(x, y)\\ &\mathbf{iff}\ \mathsf{cst}\,(\mathsf{zero})\,, \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero})) \models^\star_\rho x, y\\ &\mathbf{iff}\ \rho(x) = \mathsf{cst}\,(\mathsf{zero})\ \ \mathbf{and}\ \rho(y) = \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{zero}))\end{aligned}$$

And the second matches as follows:

$$\mathsf{cst}\,(\mathsf{zero}) \models^\star_\rho \star w, x \ \mathbf{iff}\ \rho(w) = \{\}\ \mathbf{and}\ \rho(x) = \mathsf{cst}\,(\mathsf{zero})$$ △

The abstract pattern matching semantics (Fig. 7b) is analogous, but with a few noticeable differences. First, an abstract value $\widehat{vs}$ matches a pattern p ($\widehat{vs} \mathrel{\widehat{\models}} p$) if there exists a more precise value $\widehat{vs}'$ (so $\widehat{vs}' \sqsubseteq \widehat{vs}$) and an abstract binding environment $\widehat{\rho}$ with $\mathbf{dom}\ \widehat{\rho} = \mathsf{vars}(p)$ so that $\widehat{vs}' \mathrel{\widehat{\models}_{\widehat{\rho}}} p$. The reason for using a more precise shape is the potential loss of information during over-approximation—a more precise value might have matched the pattern, even if the relaxed value does not necessarily. Second, sequences are abstracted by shape–lengths pairs, which needs to be taken into account by sequence matching rules. This is most visible in the very last rule, with a star pattern $\star x$, where we accept any assignment to a set abstraction $\widehat{vs}$ which has a more precise shape and a smaller length.

Computing pattern matches

The declarative satisfiability semantics of patterns, albeit quite clean, is unfortunately not directly computable. In Rabit, we rely on an abstract operational semantics (see **??**), translated from the concrete operational pattern matching semantics [2], using similar technique to the one presented in Sect. 5. The interesting ideas are in the refining semantic operators used, which we will discuss further.

Semantic operators with refinement Since Rascal supports non-linear matching, it becomes necessary to merge environments computed when matching sub-patterns to check whether a match succeeds or not. In abstract interpretation, we can refine the abstract environments when merging for each possibility. Consider when merging two abstract environments, where some variable x is assigned to $\widehat{vs}$ in one, and $\widehat{vs}'$ in the other. If $\widehat{vs}'$ is possibly equal to $\widehat{vs}$, we refine both values using this equality assumption

$$
\begin{array}{rll}
k(\underline{v}) \models_\rho k(\underline{p}) & \textbf{iff} & \underline{t} \text{ are parameter types of } k \\
 & & \textbf{and } \underline{v : t'} \textbf{ and } \underline{t' <: t} \\
 & & \textbf{and } \underline{v} \models^\star_\rho \underline{p} \\
v \models_\rho x & \textbf{iff} & \rho(x) = v \\
\{\underline{v}\} \models_\rho \{\underline{\star p}\} & \textbf{iff} & \underline{v} \models^\star_\rho \underline{\star p} \\
\varepsilon \models_\rho \varepsilon & \textbf{always} & \\
v, \underline{v'} \models_\rho p, \underline{\star p'} & \textbf{iff} & v \models_\rho p \textbf{ and } \underline{v'} \models^\star_\rho \underline{\star p'} \\
\underline{v}, \underline{v'} \models_\rho \star x, \underline{\star p'} & \textbf{iff} & \rho(x) = \{\underline{v}\} \textbf{ and } \underline{v'} \models^\star_\rho \underline{\star p'}
\end{array}
$$

(a) Concrete ($v \models_\rho p$ reads: v matches p with ρ)

$$
\begin{array}{rll}
k(\underline{\widehat{vs}}) \mathrel{\widehat{\models}}_{\widehat{\rho}} k(\underline{p}) & \textbf{iff} & \underline{t} \text{ are parameter types of } k \\
 & & \textbf{and } \underline{\widehat{vs} \mathrel{\widehat{:}} t'} \textbf{ and } \underline{t' \mathrel{\widehat{<:}} t} \textbf{ and } \underline{\widehat{vs}} \mathrel{\widehat{\models}}^\star_{\widehat{\rho}} \underline{p} \\
\widehat{vs} \mathrel{\widehat{\models}}_{\widehat{\rho}} x & \textbf{iff} & \widehat{\rho}(x) \sqsubseteq \widehat{vs} \\
\{\underline{\widehat{vs}}\}_{[l;u]} \mathrel{\widehat{\models}}_{\widehat{\rho}} \{\underline{\star p}\} & \textbf{iff} & \widehat{vs}, [l;u] \mathrel{\widehat{\models}}^\star_{\widehat{\rho}} \underline{\star p} \\
\widehat{vs}, [0;u] \mathrel{\widehat{\models}}_{\widehat{\rho}} \varepsilon & \textbf{always} & \\
\widehat{vs}, [l;u] \mathrel{\widehat{\models}}_{\widehat{\rho}} p, \underline{\star p'} & \textbf{iff} & u > 0 \textbf{ and } \widehat{vs} \mathrel{\widehat{\models}}_{\widehat{\rho}} p \\
 & & \textbf{and } \widehat{vs}, [l-1; u-1] \mathrel{\widehat{\models}}^\star_{\widehat{\rho}} p, \underline{\star p'} \\
\widehat{vs}, [l;u] \mathrel{\widehat{\models}}_{\widehat{\rho}} \star x, \underline{\star p'} & \textbf{iff} & \widehat{\rho}(x) = \{\widehat{vs}'\}_{[l';u']} \\
 & & \textbf{and } l' \leq l \textbf{ and } u' \leq u \textbf{ and } \widehat{vs}' \sqsubseteq \widehat{vs} \\
 & & \textbf{and } \widehat{vs}, [l-u'; u-l'] \mathrel{\widehat{\models}}^\star_{\widehat{\rho}} \underline{\star p'}
\end{array}
$$

(b) Abstract ($\widehat{vs} \mathrel{\widehat{\models}}_{\widehat{\rho}} \widehat{p}$ reads: $\widehat{vs}$ may match $\widehat{p}$ with $\widehat{\rho}$)

Figure 7. Satisfiability semantics for pattern matching

$\widehat{vs} \mathrel{\widehat{=}} \widehat{vs}'$. Here, we have that abstract equality is defined as the greatest lower bound if the value is non-bottom, i.e. $\widehat{vs} \mathrel{\widehat{=}} \widehat{vs}' \triangleq \{\widehat{vs}'' | \widehat{vs}'' = \widehat{vs} \sqcap \widehat{vs}' \neq \bot\}$. Similarly, we can also refine both values if they are possibly non-equal $\widehat{vs} \mathrel{\widehat{\neq}} \widehat{vs}'$. Here, abstract inequality is defined using relative complements:

$$
\widehat{vs} \mathrel{\widehat{\neq}} \widehat{vs}' \triangleq \begin{array}{l} \left\{(\widehat{vs}'', \widehat{vs}') | \widehat{vs}'' = \widehat{vs} \setminus (\widehat{vs} \sqcap \widehat{vs}') \neq \bot\right\} \cup \\ \left\{(\widehat{vs}, \widehat{vs}'') | \widehat{vs}'' = \widehat{vs}' \setminus (\widehat{vs} \sqcap \widehat{vs}') \neq \bot\right\} \end{array}
$$

In our abstract domains, the relative complement ($\setminus$) is limited. We heuristically define it for interesting cases, and otherwise it degrades to identity in the first argument (no refinement). There are however useful cases, e.g., for excluding unary constructors $\mathsf{suc}\,(\mathsf{Nat}) \wr \mathsf{zero} \setminus \mathsf{zero} = \mathsf{suc}\,(\mathsf{Nat})$ or at the end points of a lattice $[1;10] \setminus [1;2] = [3;10]$.

Similarly, for matching against a constructor pattern $k(\underline{p})$, the core idea is that we should be able to partition our value space into two: the abstract values that match the constructor and those that do not. For those values that possibly match $k(\underline{p})$, we produce a refined value with k as the only choice, making sure that the sub-values in the result are refined by the sub-patterns $\underline{p}$.

Otherwise, we try to exclude k from the refined value. For a data type abstraction exclusion removes the pattern constructor from the possible choices

$$
\widehat{\mathsf{exclude}}(k(\widehat{vs}) \wr k_1(\widehat{vs}_1) \wr \cdots \wr k_n(\widehat{vs}_n), k) = k_1(\widehat{vs}_1) \wr \cdots \wr k_n(\widehat{vs}_n)
$$

and otherwise it will not change the input shape.

7 Traversals

First-class traversals are a key feature of high-level transformation languages, since they enable effectively transforming large abstract syntax trees. In Rascal, this is implemented using the visit expression visit e $\underline{cs}$. We will focus on the rules of bottom-up traversals, but the challenges and properties are common for all strategies supported in Rascal.

Bottom-up Traversal The core idea of a bottom-up traversal of an abstract value $\widehat{vs}$, is to first traverse children of the value $\widehat{\mathsf{children}}(\widehat{vs})$ possibly rewriting them, then reconstruct (using $\widehat{\mathsf{recons}}$) a new value using the rewritten children and finally traversing the reconstructed value. The main challenging step is handling traversal of children, whose representation and thus execution rules depend on the particular abstract value.

Concretely, the $\widehat{\mathsf{children}}(\widehat{vs})$ function returns a pair $(\widehat{vs}', \widehat{cvs})$ where the first component $\widehat{vs}'$ is a refinement of $\widehat{vs}$ that matches the shape of children $\widehat{cvs}$ in the second component. For data type values the representation of children is a heterogeneous sequence of abstract values $\underline{\widehat{vs}''}$, while for set values (and the top element) the representation of children is a pair $(\widehat{vs}'', [l;u])$ with the first component representing the shape of elements and the second representing their count. For example,

$$
\widehat{\mathsf{children}}(\mathsf{mult}\,(\mathsf{Expr}, \mathsf{Expr}) \wr \mathsf{cst}\,(\mathsf{suc}\,(\mathsf{Nat}))) = \left\{ \begin{array}{l} (\mathsf{mult}\,(\mathsf{Expr}, \mathsf{Expr}), (\mathsf{Expr}, \mathsf{Expr})), \\ \quad (\mathsf{cst}\,(\mathsf{suc}\,(\mathsf{Nat})), \mathsf{suc}\,(\mathsf{Nat})) \end{array} \right\}
$$

and

$$
\widehat{\mathsf{children}}(\{\mathsf{Expr}\}_{[1;10]}) = \{(\{\mathsf{Expr}\}_{[1;10]}, (\mathsf{Expr}, [1;10]))\}
$$

Note how the $\widehat{\mathsf{children}}$ function partitions the alternatives for data-types, in order to preserve precision when traversing each corresponding sequence of value shapes for the children.

Traversing children As previously mentioned, the shape of execution rules depend on the representation of children; this is consistent with the requirements imposed by

Schmidt [45]. For heterogeneous sequences of value shapes $\widehat{vs}$ (as produced by constructors), the execution rules iterate through the sequence recursively traversing each element (ABUC-E, ABUC-M). We use the $\widehat{\text{vcombine}}$ abstract function to combine the result of traversing a single element with the result of traversing the rest of the children (see ABUC-M), producing a success result if any rewriting was applied and otherwise producing a failure, refining the shape of the children. Due to over-approximation we may re-traverse the same or a more precise value on recursion, and so we need to use trace memoization (Sect. 8) to terminate. For example the children of an expression Expr refer to itself:

$$\widehat{\text{children}}(\text{Expr}) = \left\{ \begin{array}{c} (\text{mult}(\text{Expr}, \text{Expr}), (\text{Expr}, \text{Expr})), \\ (\text{cst}(\text{Nat}), \text{Nat}), (\text{var}(\text{str}), \text{str}) \end{array} \right\}$$

Traversing children represented by a shape-length pair, is directed by the length interval $[l; u]$. If 0 is a possible value of the length interval, then traversal can finish, refining the input shape to be empty (ABUS-E). Otherwise, we perform another traversal recursively on the shape of elements and recursively on a new shape-length pair which decreases the length, finally combining their values (ABUS-M). Note, that if the length is unbounded, e.g. $[0; \infty]$, then the value can be decreased forever and trace memoization is also needed here for termination. This means that trace memoization must here be nested breadth-wise (when recursing on an unbounded sequence of children), in addition to depth-wise (when recursing on children); this can be computationally expensive, and we will discuss in Sect. 9 how our implementation handles that.

Evaluating Cases During traversal, the target value will be rewritten with a sequence of cases. The evaluation of a case sequence is straight-forward, iterating through the possible cases, pattern matching against each pattern and executing the corresponding expression when applicable. The main idea to notice, is that when the abstract value fails to match a pattern, the refined value is used to match against the rest of the cases (ACS-M-F). This ensures that the order of patterns influences the refinement, and we get a more precise abstract shape that better matches the set of concrete shapes during execution.

8 Trace Memoization

Abstract interpretation and static program analysis in general perform fixed-point calculation for analysing unbounded loops and recursion. In Schmidt-style abstract interpretation [45], the main technique to handle recursion is *trace memoization*. The core idea of *trace memoization* is to detect non-structural re-evaluation of the same program element—i.e., when the evaluation of a program element is recursively dependent on the evaluation of itself, like a while-loop or traversal. In such case, widening on the input values and state produces an over-approximating finite, circular dependency in the trace of the program element, which can be solved by classical fixed-point iteration.

The trace memoization strategies suggested by existing work, do however not directly apply to the constructs present in Rascal. The technique by Schmidt [45] only works for constructs like while-loops where the self-recursive evaluation happens in a tail position. Rosendahl [43] shows how to support more general recursive dependencies, but only for finite input domains, whereas our input domains are all infinite.

We have therefore further extended the trace memoization strategy suggested by Rosendahl [43] to work with abstract inputs from infinite domains. The extension is still terminating, sound and, additionally, it allows calculating results with good precision. The core idea is to partition the infinite input domain using a finite domain of elements, and on recursion degrade input values using previously met input values from the same partition. We assume that all our domains are lattices with a widening operator. Consider a recursive operational semantics judgment $i \Longrightarrow o$, with i being an input from domain $\widehat{\text{Input}}$, and o being the output from domain $\widehat{\text{Output}}$. For this judgment, we associate a memoization map $\widehat{M} \in \widehat{\text{PInput}} \to \widehat{\text{Input}} \times \widehat{\text{Output}}$ where $\widehat{\text{PInput}}$ is a finite partitioning domain that has a Galois connection with our actual input, i.e. $\widehat{\text{Input}} \underset{\alpha_{\widehat{PI}}}{\overset{\gamma_{\widehat{PI}}}{\leftrightarrows}} \widehat{\text{PInput}}$. The memoization map keeps track of the previously seen input and corresponding output for values in the partition domain. For example, for input from our value domain $\widehat{\text{Value}}$ we can use the corresponding type from the domain Type as input to the memoization map.[2] So for values 1 and $[2; 3]$ we would use int, while for mult(Expr, Expr) we would use the defining data type Expr.

We perform a fixed-point calculation over the evaluation of input i. Initially, the memoization map $\widehat{M}$ is $\lambda pi.(\bot, \bot)$, and during evaluation we check whether there was already a value from the same partition as i, i.e., $\alpha_{\widehat{PI}}(i) \in \text{dom } \widehat{M}$. At each iteration, there are then two possibilities:

Hit The corresponding input partition key is in the memoization map and a less precise input is stored, so $\widehat{M}(\alpha_{\widehat{PI}}(i)) = (i', o')$ where $i \sqsubseteq_{\widehat{\text{Input}}} i'$. Here, the output value o that is stored in the memoization map is returned as result.

Widen The corresponding input partition key is in the memoization map, but an unrelated or more precise input is stored, i.e., $\widehat{M}(\alpha_{\widehat{PI}}(i)) = (i'', o'')$ where $i \not\sqsubseteq_{\widehat{\text{Input}}} i''$. In this case we continue evaluation but with a widened input $i' = i'' \nabla_{\widehat{\text{Input}}} (i'' \sqcup i)$ and an updated map $\widehat{M}' = [\alpha_{\widehat{PI}}(i) \mapsto (i', o_{\text{prev}})]$. Here, o_{prev} is the output of the last iteration for the fixed-point calculation for input i', and is assigned $\bot$ on the initial iteration.

[2] Provided that we put a fixed-bound on the depth of type parameters of collections.

The presented trace memoization technique is systematically applicable to the traversal rules from Section 7 and similar infinite derivations. This suggests that the presented trace memoization technique can be automated for operational semantics rules of certain shapes, extending the technique presented by [9].

Intuitively, the technique is terminating because the partitioning is finite, and widening ensures that we reach an upper bound of possible inputs in a finite number of steps, eventually getting a hit. The fixed-point iteration also uses widening to calculate an upper bound, which similarly finishes in a number of steps. The technique is sound because we only use output for previous input that is less precise. This ensures that our function is continuous and that a fixed-point exists.

9 Experimental Evaluation

We want to show that the presented technique works on a variety of transformations in Rascal. We shall consider the following research questions:

RQ1 Is it possible to abstractly interpret realistic Rascal programs using type and inductive shape analysis to verify interesting properties?

RQ2 Capturing what relevant properties may require more than inductive shapes, and what extensions to the technique would be needed?

We demonstrate the ability of Rabit to verify type and shape properties, using five transformation programs across various applications. Three programs are classic examples, and two are extracted from open source projects.

Negation Normal Form (**NNF**) transformation [27, Section 2.5] is a classical rewrite of a propositional formula to combination of conjunctions and disjunctions of literals, so negations appear only next to atoms. An implementation of this transformation should guarantee the following:

P1 Implication is not used as a connective in the result
P2 All negations in the result are in front of atoms

Rename Struct Field (**RSF**) refactoring changes the name of a field in a struct, and that all corresponding field access expressions are renamed correctly as well:

P3 Structure should not define a field with the old field name
P4 No field access expression to the old field

Desugar Oberon-0 (**DSO**) transformation, translates for-loops and switch-statements to while-loops and nested if-statements, respectively. The transformation is part of the Oberon-0 [54] implementation in Rascal [6], containing all the necessary stages for compiling a structured imperative programming language.

P5 for should be correctly desugared to while
P6 switch should be correctly desugared to if
P7 No auxiliary data in output

Code Generation for Glagol (**G2P**) a DSL for REST-like web development, translated to PHP for execution.[3] We are interested in the part of the generator that translates Glagol expressions to PHP, and the following properties:

P8 Output only simple PHP expressions for simple Glagol expression inputs
P9 No unary PHP expressions if no sign marks or negations in Glagol input

Mini Calculational Language (**MCL**) a programming language text-book [47] implementation of a small expression language, with arithmetic and logical expressions, variables, if-expressions, and let-bindings. The implementation contains an expression simplifier (larger version of running example in Fig. 2), a type inference procedure, an interpreter and a compiler.

P10 Simplification procedure produces a simplified expression with no additions with 0, multiplications with 1 or 0, subtractions with 0, logical expressions with constant Boolean operands, and if-expressions with constant Boolean conditions.
P11 Arithmetic expressions with no variables have type int and no type errors
P12 Interpreting expressions with no integer constants and let's gives only Boolean values
P13 Compiling expressions with no if's produces no goto'sand if instructions
P14 Compiling expressions with no if's produces no labels and does not change label counter

All these transformations satisfy the following criteria:

1. They are formulated by an independent source,
2. They can be translated in relatively straightforward manner to our subject of Rascal, and
3. They exercise important constructs, including visitors and the expressive pattern matching

We have ported all these programs to Rascal Light.

Threats to validity. The programs are not selected randomly, thus it is hard to generalize the results for other transformations. We mitigated this by selecting transformations that are realistic and vary in authors, programming style and purpose. While translating the programs to Rascal Light, we strived to minimize the amount of changes, but generally bias cannot be ruled out entirely.

Implementation. We have implemented the abstract interpreter in a prototype tool, Rabit, for all of Rascal Light following the process described in sections 5 to 8. This required handling additional aspects, not discussed in the paper:

1. possibly undefined values (■)
2. extended result state with more control flow constructs, backtracking, exceptions, loop control, and

[3] https://github.com/BulgariaPHP/glagol-dsl

3. fine-tuning memoization strategies to the different looping constructs and recursive calls

The input to the tool is completely specified by the user; by default, we use the top element $\top$ for the types specified as input. The user can specify the initial data-type refinements, store and parameters, to get a more precise result for target function to be abstractly interpreted. The output of the tool is the abstract result value set of abstractly interpreting target function, the resulting store state and the set of relevant inferred data-type refinements.

The paper primarily demonstrates the abstract interpretation technique that computes this approximation of the output values. The target properties must be checked manually by the user against the inferred abstract value shape. A potential future work is to wrap Rabit in a more user-oriented interface. This interface could allow some logic to easily specify target properties and a checker for such logic on the inferred shapes.

In addition to trace memoization, Rabit uses caches executions for performance reasons. This is especially important in nested fixed-point iterations, where the same expression might be re-executed in the same state and it would spend time recomputing the same result as in the previous iteration.

The implementation extends standard regular tree grammar operations [1, 18], to handle the recursive equations for the expressive abstract domains, including base values, collections and heterogeneous data types. We use a more precise partitioning strategy for trace memoization when needed, which also takes the set of available constructors into account for data types. The source code of our implementation, including subject transformations, is freely available. [4]

[4] Omitted for blinding.

Transformation	LOC	Runtime [s]	Property	Verified
NNF	15	7.3	P1	✓
			P2	✓
RSF	35	6.0	P3	✗
			P4	✓
DSO	125	25.0	P5	✓
			P6	✓
			P7	✗
G2P	350	1.6	P8	✓
		3.5	P9	✓
MCL	298	1.6	P10	✓
		0.7	P11	✓
		0.6	P12	✓
		0.9	P13	✓
			P14	✓

Table 1. Time and success rate for analyzing programs and properties presented earlier this section. Time is the median of five runs. If the same time is reported for multiple properties, then they could be verified on the same input

```
data FormulaIn = and(FormulaIn, FormulaIn)
               | atom(str) | neg(FormulaIn)
               | imp(FormulaIn, FormulaIn)
               | or(FormulaIn, FormulaIn)

data FormulaOut = and(FormulaOut, FormulaOut)
                | atom(str) | neg(atom(str))
                | or(FormulaOut, FormulaOut)
```

Figure 8. Initial and inferred refinement types for NNF

Results. We ran the experiments using Scala 2.12.2 on a 2012 Core i5 MacBook Pro. Table 1 summarizes the size of the programs, the runtime of the abstract interpreter, and whether the properties have been verified. We remark that all programs use the high-level expressive features of Rascal and are thus significantly more succinct than comparable code in general purpose languages. The runtime, varying from single seconds to less than a minute, is reasonable. All, but two, properties were successfully verified.

Lines 1–2 in Fig. 9 show the input refinement type FormulaIn for the normalization procedure. Note the presence of implication, and free use of negation at any level. For this example, the inferred inductive output type FormulaOut (lines 4–5) specifies that the implication is not present in the output (P1), and negation only allows atoms as subformulae (P2). In fact, Rabit inferred a precise characterization of negation normal form as an inductive data type.

10 Related Work

Data and shape domains. We start with discussing techniques that could be used to make Rabit infer more precise shapes and verify properties like P3 and P7.

To verify P3, we need to be able to relate field names to their corresponding definitions in the field definition map of a class, which is not possible using the presented non-relational abstract domains. Relational abstract interpreteration [34] allows specifying such constraints that relate values across different variables, and even inside and across substructures [14, 26, 32]. Furthermore, they allow inferring properties that are non-local, e.g., that two values not only have similar shape but are exactly equal.

For a concrete input of P7, we know that the number of auxiliary data elements decreases on each iteration, but this information is lost in our abstraction of data structures. A possible solution could be to allow *abstract attributes* that extract additional information about the abstracted structures. For P7, a generalization of the multiset abstraction [37] for data types, could be useful to track e.g., the auxiliary statement count, and show that they decrease using multiset-ordering [22] like in term rewriting. Other abstract attributes [10, 38, 50] include set of contained elements and size of data type values.

Other techniques [4, 14, 52] support inferring inductive relational properties for general data-types—such as the binary tree property—but require a pre-specified structure specifying the possible places relational refinement could happen. They all present useful extension direction. To the best of our knowledge, none of these, has been applied to transformation programs.

Modular program analysis. Cousot and Cousot [19] present a general framework for modularly constructing program analyses, but it requires a language with a compositional control flow which Rascal does not have. The framework suggests using symbolic relational domains for domain composition, but it is generally undecidable to infer inductive properties for arbitrary sets of constraints [36]. Toubhans, Rival and Chang [42, 51] develops the ideas of modular domain design for pointer-manipulating programs supporting a rich set of fixed data abstractions, whereas our domain construction focuses on providing automated inference of inductive refinement types based on pure heterogeneous data-structures (algebraic data types, collections, and basic values). Definitional interpreters have been suggested as a technique for building compositional abstract interpreters [21]. The idea is to rely on a monad transformer stack to share the implementation of the concrete and abstract interpreters, and parametrize over the semantic operations which are given different semantics depending on the mode of execution (concrete, abstract). We believe that our interpreter could have benefited from being written in such style[5], which would have complemented our modular domain construction well. To ensure termination they rely on a *caching* algorithm which is similar to ordinary finite input *trace memoization* [43], which have the same limitations that our generalized *partition-driven trace memoization* resolves in order to get a useful analyses for a complex language like Rascal.

Analysis of pattern matching definitions. Garrigue [24, 25] presents algorithms for typing pattern matching on polymorphic variant types in OCaml, where the set of constructors for a data type is not fixed in advance. The theory is useful since it supports inferring simple recursive shapes of programs, but it has its limitations. The inference is syntactic and exact, which means that it will not succeed in inferring a type for many useful program (which the compiler will then reject). It is also unclear how to generalize the inference procedure to work with the rich pattern matching constructs and heterogeneous visitors present in high-level transformation languages.

Haskell supports analysing coverage of it's pattern matching language, that includes generalized algebraic data types (GADTs) and Boolean constraints [29]. It uses abstract interpretation to check coverage of constructor patterns, and an SMT solver to resolve constraints on basic values (e.g. integers). While general Haskell function calls can occur in the Boolean constraints, the analysis treats them shallowly as function symbols. This entails that some covering pattern matches, that depend on the particular semantics of the called functions, will be marked falsely as non-exhaustive.

Modern SMT solvers supports reasoning with inductive functions defined over algebraic data-types [39]. The properties they can verify are very expressive, and include inductive semantic properties. The exact solving techniques SMT solvers employ is however not very scalable, and encoding a complex transformation directly would not finish verifying even simple properties within reasonable time. Comparatively, Rabit only supports inductive shape properties, but can run within seconds for even the largest transformations in our evaluation.

Possible constructor analysis[5] has been used to calculate the actual dependencies of a predicate and make flow-sensitive analyses more precise. This is a type of shape analysis works with complex data-types and arrays. The analysis however only aims to capture the prefix of the target structures, whereas ours infer inductive refinements.

Verification of transformations. Several approaches [12, 28] for verifying model transformations encode transformations as declarative formulae for black-box automated solvers (e.g., SMT, CLP), but they only work for transformations of limited expressiveness and the reliance on solvers makes them hard to extend. Techniques for model transformation verification based on static analysis [20] scale to more expressive transformation features, but are currently focused on verification of rule errors based on types and undefinedness.

Symbolic execution has previously been suggested [3] as a way to validate high-level transformation programs. However, that work targets test generation rather than verification of properties and is only demonstrated on a toy language, whereas Rabit allows static analysis of a significant subset of Rascal and evaluated on realistic transformations. Semantic typing [8, 13] has been used to infer recursive type and shape properties for language with high-level constructs for querying and iteration. The languages considered are small calculi compared to the supported subset of Rascal we consider, our domain and abstract interpreter design is more extensible, and our evaluation is extensive showing that our works well with realistic Rascal programs.

Translation validation [40] has been used in conjunction with symbolic transfer functions to verify equivalence between a program annotated with abstract invariants and its translated assembly output. The technique is complementary to ours, since it can verify rich semantic properties but only for concrete input, while our technique verifies simpler properties as shapes but for all possible inputs.

[5] We only learned about this related work at a late stage

11 Conclusion

Our goal was to use abstract interpretation to give a solid semantic foundation for analyzing programs in modern high-level transformation languages. To this end we have designed and formalized a Schmidt-style abstract interpreter, including and an extension of *trace memoization* to infinite input domains (previously only known to work for finite domains). The proposed modular construction of abstract domains was vital for handling a language of this scale and complexity, including collections and heterogeneous data types.

We implemented the interpreter as a tool, Rascal ABstract Interpretation Tool (Rabit). Rabit supports a non-trivial subset of Rascal, containing key features: several traversal strategies, expressive pattern matching, backtracking, exceptions and control operators, and generalized looping constructs. We evaluated Rabit on classical transformations and on examples selected from open source projects, showing it allows verification of a series of sophisticated type and shape properties for these transformations.

References

[1] Alexander Aiken and Brian R. Murphy. 1991. Implementing Regular Tree Expressions. In *FPLCA 1991*. 427–447. https://doi.org/10.1007/3540543961_21

[2] Ahmad Salim Al-Sibahi. 2017. The Formal Semantics of Rascal Light. *CoRR* abs/1703.02312 (2017). http://arxiv.org/abs/1703.02312

[3] Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wasowski. 2016. Symbolic execution of high-level transformations. In *SLE 2016*. 207–220. http://dl.acm.org/citation.cfm?id=2997382

[4] Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. 2015. Spatial Interpolants. In *ESOP 2015*. 634–660. https://doi.org/10.1007/978-3-662-46669-8_26

[5] Oana Fabiana Andreescu, Thomas Jensen, and Stéphane Lescuyer. 2015. Dependency Analysis of Functional Specifications with Algebraic Data Structures. In *ICFEM 2015*. 116–133. https://doi.org/10.1007/978-3-319-25423-4_8

[6] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. 2015. Modular language implementation in Rascal - Experience Report. *Sci. Comput. Program.* 114 (2015), 7–19. http://dx.doi.org/10.1016/j.scico.2015.11.003

[7] Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. 10, 4 (2003), 265–289.

[8] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. 2013. Static and dynamic semantics of NoSQL languages. In *POPL 2013*. 101–114. https://doi.org/10.1145/2429069.2429083

[9] Martin Bodin, Thomas Jensen, and Alan Schmitt. 2015. Certified Abstract Interpretation with Pretty-Big-Step Semantics. In *CPP 2015*. 29–40. https://doi.org/10.1145/2676724.2693174

[10] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2012. Abstract Domains for Automated Reasoning about List-Manipulating Programs with Infinite Data. In *VMCAI 2012*. 1–22. https://doi.org/10.1007/978-3-642-27940-9_1

[11] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* 72, 1-2 (2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[12] Fabian Büttner, Marina Egea, and Jordi Cabot. 2012. On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In *MODELS 2012*. 432–448. https://doi.org/10.1007/978-3-642-33666-9_28

[13] Giuseppe Castagna and Kim Nguyen. 2008. Typed iterators for XML. In *ICFP 2008*. 15–26. https://doi.org/10.1145/1411204.1411210

[14] Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational Inductive Shape Analysis. In *POPL 2008*. 247–260. https://doi.org/10.1145/1328438.1328469

[15] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *ICFP 2010*. 3–14. https://doi.org/10.1145/1863543.1863547

[16] James R. Cordy. 2006. The TXL source transformation language. *Sci. Comput. Program.* 61, 3 (2006), 190–210. https://doi.org/10.1016/j.scico.2006.04.002

[17] Patrick Cousot. 2003. Verification by Abstract Interpretation. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*. 243–268. https://doi.org/10.1007/978-3-540-39910-0_11

[18] Patrick Cousot and Radhia Cousot. 1995. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *FPCA 1995*. 170–181. http://doi.acm.org/10.1145/224164.224199

[19] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *CC 2002*. 159–178. https://doi.org/10.1007/3-540-45937-5_13

[20] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2017. Static Analysis of Model Transformations. *IEEE Trans. Software Eng.* 43, 9 (2017), 868–897. https://doi.org/10.1109/TSE.2016.2635137

[21] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

[22] Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Commun. ACM* 22, 8 (1979), 465–476. https://doi.org/10.1145/359138.359142

[23] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *PLDI 1991*. 268–277. http://doi.acm.org/10.1145/113445.113468

[24] Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13.

[25] Jacques Garrigue. 2004. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages*.

[26] Nicolas Halbwachs and Mathias Péron. 2008. Discovering properties about arrays in simple programs. In *PLDI 2008*. 339–348. https://doi.org/10.1145/1375581.1375623

[27] John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

[28] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. 2011. Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In *MODELS 2011*. 653–667. https://doi.org/10.1007/978-3-642-24485-8_48

[29] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. GADTs meet their match: pattern-matching warnings that account for GADTs, guards, and laziness. In *ICFP 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 424–436. https://doi.org/10.1145/2784731.2784748

[30] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels* (1 ed.). Addison-Wesley Professional.

[31] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2011. EASY Meta-programming with Rascal. In *GTTSE III*, JoãoM. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). 222–289. https://doi.org/10.1007/978-3-642-18023-1_6

[32] Jiangchao Liu and Xavier Rival. 2017. An array content static analysis based on non-contiguous partitions. *Computer Languages, Systems & Structures* 47 (2017), 104–129. https://doi.org/10.1016/j.cl.2016.01.005

[33] Neil Mitchell and Colin Runciman. 2007. Uniform boilerplate and list processing. In *Haskell 2007, Freiburg, Germany*. 49–60. https:

//doi.org/10.1145/1291201.1291208

[34] Alan Mycroft and Neil D. Jones. 1985. A relational framework for abstract interpretation. In *Programs as Data Objects*. 156–171. https://doi.org/10.1007/3-540-16446-4_9

[35] F. Nielson, H. Nielson, and C. Hankin. 1999. *Principles of Program Analysis*. Springer.

[36] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. 2016. Decidability of inferring inductive invariants. In *POPL 2016*. 217–231. https://doi.org/10.1145/2837614.2837640

[37] Valentin Perrelle and Nicolas Halbwachs. 2010. An Analysis of Permutations in Arrays. In *VMCAI 2010*. 279–294. https://doi.org/10.1007/978-3-642-11319-2_21

[38] Tuan-Hung Pham and Michael W. Whalen. 2013. An Improved Unrolling-Based Decision Procedure for Algebraic Data Types. In *VSTTE 2013*. 129–148. https://doi.org/10.1007/978-3-642-54108-7_7

[39] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *VMCAI 2015*. 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

[40] Xavier Rival. 2004. Symbolic transfer function-based approaches to certified compilation. In *POPL 2004*. 1–13. https://doi.org/10.1145/964001.964002

[41] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26. https://doi.org/10.1145/1275497.1275501

[42] Xavier Rival, Antoine Toubhans, and Bor-Yuh Evan Chang. 2014. Construction of Abstract Domains for Heterogeneous Properties. In *ISoLA 2014*. 489–492. https://doi.org/10.1007/978-3-662-45231-8_40

[43] Mads Rosendahl. 2013. Abstract Interpretation as a Programming Language. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*. 84–104. https://doi.org/10.4204/EPTCS.129.7

[44] John M. Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Trans. Software Eng.* 24, 9 (1998), 709–720. https://doi.org/10.1109/32.713327

[45] David A. Schmidt. 1998. Trace-Based Abstract Interpretation of Operational Semantics. *Lisp and Symbolic Computation* 10, 3 (1998), 237–271.

[46] Dana S. Scott. 1976. Data Types as Lattices. *SIAM J. Comput.* 5, 3 (1976), 522–587. http://dx.doi.org/10.1137/0205037

[47] Peter Sestoft and Niels Hallenberg. 2017. *Programming language concepts*. Springer.

[48] AnthonyM. Sloane. 2011. Lightweight Language Processing in Kiama. In *GTTSE III*, JoãoM. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Lecture Notes in Computer Science, Vol. 6491. Springer Berlin Heidelberg, 408–425. https://doi.org/10.1007/978-3-642-18023-1_12

[49] Michael B. Smyth and Gordon D. Plotkin. 1982. The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.* 11, 4 (1982), 761–783. http://dx.doi.org/10.1137/0211062

[50] Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision procedures for algebraic data types with abstractions. In *POPL 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 199–210. https://doi.org/10.1145/1706299.1706325

[51] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. 2013. Reduced Product Combination of Abstract Domains for Shapes. In *VMCAI 2013*. 375–395. https://doi.org/10.1007/978-3-642-35873-9_23

[52] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *ESOP 2013*. 209–228. https://doi.org/10.1007/978-3-642-37036-6_13

[53] Glynn Winskel. 1993. Information Systems. MIT Press, Chapter 12.

[54] Niklaus Wirth. 1996. *Compiler Construction*. Addison-Wesley.

[55] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *PLDI 1998*. 249–257. https://doi.org/10.1145/277650.277732

A Operational Pattern Matching

Computing Pattern Matching The judgements are presented in Fig. 8 for both the concrete and abstract rules. Consider the concrete (top-left) judgement: a value v matches a pattern p, given a store σ, producing a sequence of binding environments $\underline{\rho}$. The binding environments form a sequence, since multiple concrete environments, say ρ_1 and ρ_2, can make v match against p, i.e., $v \models_{\rho_1} p$ and $v \models_{\rho_2} p$. Backtracking using the fail-expression, allows the programmer to explore a different assignment from the sequence of environments, until no possible assignment is left.

For an ordinary pattern p (top) the abstraction relation is direct: an abstract store $\widehat{\sigma}$ abstracts a concrete store σ and a value shape $\widehat{vs}$ abstracts a concrete value v. The notable change is that the abstract semantics uses a set of abstract binding environments $\widehat{\varrho} \subseteq \widehat{\mathsf{Store}} \times \widehat{\mathsf{ValueShape}} \times \widehat{\mathsf{BindingEnv}}_{\perp}$ that not only abstracts over the sequence of concrete binding environments $\underline{\rho}$, but also, for each abstract binding environment stores the corresponding refinement of the input abstract store $\widehat{\sigma}$ and the corresponding refinement of the matched value shape $\widehat{vs}$ according to the matched pattern.

For sequences of set sub-patterns $\underline{\star p}$, the sequence of concrete values $\underline{v}$ is abstracted by two components: the shape of values $\widehat{vs}$ and an interval approximating the length of the value sequence $[l; u]$. Both of these values are refined as a result of the matching, which is captured by the abstract binding environment $\widehat{\varrho}$ (of the same type as for the simple patterns), since we treat the value refined as the abstract set containing the values of the given shape and of given cardinality. The concrete semantics of set sub-patterns also contains a backtracking state $\mathbb{V}$ which is not used in the abstract semantics, because the abstraction of set elements is coarse and we thus abstractly consider all possible subset assignments at the same time (joining instead of backtracking).

Operational Rules We will show how refinement is calculated by the abstract operational semantics by presenting some of key rules for abstract pattern matching. Rascal also allows non-linear pattern matching against assigned store variables, and it is possible to use this information for refining the input store and abstract value. In the AP-V-U rule we match the variable to the value shape and restrict the shape abstraction for the variable value to match the pattern. The binding environment does not change as the name is already bound in the store. In the AP-V-F rule, the matching fails ($\perp$), and then we learn that the value shape in the store should be refined to something that does not match.

$$\text{AP-V-U}\ \frac{\widehat{\sigma}(x) = (b, \widehat{vs}') \qquad \widehat{vs}' \neq \perp_{\widehat{\mathrm{VS}}} \qquad \widehat{vs}'' \in (\widehat{vs} \,\widehat{=}\, \widehat{vs}') \qquad \widehat{\sigma}' = \widehat{\sigma}[x \mapsto (\mathrm{ff}, \widehat{vs}'')]}{\widehat{\sigma} \vdash x \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}', \widehat{vs}'', [])}$$

$$\text{AP-V-F}\ \frac{\widehat{\sigma}(x) = (b, \widehat{vs}') \qquad \widehat{vs}' \neq \perp_{\widehat{\mathrm{VS}}} \qquad (\widehat{vs}'', \widehat{vs}''') \in (\widehat{vs} \,\widehat{\neq}\, \widehat{vs}') \qquad \widehat{\sigma}' = \widehat{\sigma}[x \mapsto (\mathrm{ff}, \widehat{vs}''')]}{\widehat{\sigma} \vdash x \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}', \widehat{vs}'', \perp)}$$

We also show the AP-V-B (abstract pattern-variable-bind) rule which simply binds the variable in the binding environment, assuming that it is possibly not assigned in the store (a free name).

$$\text{AP-V-B}\ \frac{\widehat{\sigma}(x) = (\mathrm{tt}, \widehat{vs}')}{\widehat{\sigma} \vdash x \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}[x \mapsto (\mathrm{tt}, \perp_{\widehat{\mathrm{VS}}})], \widehat{vs}, [x \mapsto \widehat{vs}])}$$

If our matched abstract value possibly contains the pattern constructor k (AP-C-S rule: abstract pattern-constructor-success) we produce an abstract value with k containing the sub-values refined against constructor sub-patterns:

$$\text{AP-C-S}\ \frac{\begin{array}{c}\mathsf{data}\ at = \cdots \mid k(\underline{t}) \mid \ldots \\ (\text{success}\ k(\underline{\widehat{vs}'})) \in \widehat{\mathsf{unfold}}(\widehat{vs}, at) \\ \widehat{\sigma} \vdash p_1 \overset{?}{\coloneqq} \widehat{vs}'_1 \xRightarrow[\text{a-match}]{} \widehat{\varrho}_1 \ldots \widehat{\sigma} \vdash p_{\mathrm{n}} \overset{?}{\coloneqq} \widehat{vs}'_{\mathrm{n}} \xRightarrow[\text{a-match}]{} \widehat{\varrho}_{\mathrm{n}} \\ (\widehat{\sigma}'_1, \widehat{vs}'_1, \widehat{\rho}^{?}_1) \in \widehat{\varrho}_1 \ldots (\widehat{\sigma}'_{\mathrm{n}}, \widehat{vs}'_{\mathrm{n}}, \widehat{\rho}^{?}_{\mathrm{n}}) \in \widehat{\varrho}_{\mathrm{n}}\end{array}}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-cons}]{} (\textstyle\bigsqcap_i \widehat{\sigma}_i, k(\underline{\widehat{vs}'}), \widehat{\mathsf{merge}}(\underline{\widehat{\rho}^{?}}))}$$

The total function $\widehat{\mathsf{merge}}$ unifies assignments from two binding environments point-wise by names, taking the greatest lower bound of shapes to combine bindings for a name. It yields bottom for the entire result if at least one of the point-wise meets

yields bottom (shapes for at least one name are not reconcilable). Otherwise, we try to refine the matched value to exclude the pattern constructor in the AP-C-F rules:

$$\text{AP-C-F1}\ \frac{\begin{array}{c}\mathsf{data}\ at = \cdots \mid k(\underline{t}) \mid \ldots \\ (\mathsf{success}\ k'(\underline{\widehat{vs}'})) \in \widehat{\mathsf{unfold}}(\widehat{vs}, at) \qquad k' \neq k\end{array}}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-cons}]{} (\widehat{\sigma}, \widehat{\mathsf{exclude}}(\widehat{vs}, k), \bot)}$$

$$\text{AP-C-F2}\ \frac{\mathsf{data}\ at = \cdots \mid k(\underline{t}) \mid \ldots \qquad \mathsf{error} \in \widehat{\mathsf{unfold}}(\widehat{vs}, at)}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-cons}]{} (\widehat{\sigma}, \widehat{\mathsf{exclude}}(\widehat{vs}, k), \bot)}$$

For set patterns, the refinement happens by pattern matching set sub-patterns.

$$\text{AP-S-S}\ \frac{\begin{array}{c}\mathsf{success}\ \{\widehat{vs}'\}_{[l;u]} \in \widehat{\mathsf{unfold}}(\widehat{vs}, \mathsf{set}\langle\mathsf{value}\rangle) \\ \widehat{\sigma} \vdash \underline{\star p} \overset{?}{\coloneqq} \widehat{vs}, [l;u] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho}\end{array}}{\widehat{\sigma} \vdash \{\underline{\star p}\} \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match-set}]{} \widehat{\varrho}}$$

For example, when it is possible that the abstracted value sequence $(\widehat{vs}, [l;u])$ is empty $(l = 0)$ and patterned matched against an empty set sub-pattern sequence, we can refine the result to be the empty abstract set $\{\bot\}_0$ (rule APL-E-B).

$$\text{APL-E-B}\ \frac{l \leq u \qquad l = 0}{\widehat{\sigma} \vdash \varepsilon \overset{?}{\coloneqq} \widehat{vs}, [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} (\widehat{\sigma}, \{\bot_{\widehat{\mathrm{VS}}}\}_0, \{[]\})}$$

A more complex example is the one where we try to pattern match a potentially non-empty value sequence against a set sub-pattern sequence $p, \underline{\star p'}$ starting with an ordinary pattern (APL-M-P). Here we pattern match against p and the rest of the sequence $\underline{\star p'}$ and combine the refined results of these matches producing a refinement of the containing set value by combining the refined shapes and increasing the refinement of the length by the set sub-pattern sequence by one.

$$\text{APL-M-P}\ \frac{\begin{array}{c} l \leq u \qquad u \neq 0 \qquad \widehat{\sigma} \vdash p \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}' \\ \widehat{\sigma} \vdash \underline{\star p} \overset{?}{\coloneqq} \widehat{vs}, [l-1; u-1] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}'' \\ (\widehat{\sigma}', \widehat{vs}', \widehat{\varrho}') \in \widehat{\varrho_R}' \qquad (\widehat{\sigma}'', \{\widehat{vs}''\}_{[l'';u'']}, \widehat{\varrho}'') \in \widehat{\varrho_R}'' \\ \widehat{\varrho_R}''' = \left\{ \begin{array}{l} (\widehat{\sigma}' \sqcap \widehat{\sigma}'', \{\widehat{vs}' \sqcup \widehat{vs}''\}_{[l''+1, u''+1]}, \\ \widehat{\mathsf{merge}}(\widehat{\varrho}', \widehat{\varrho}'')) \end{array} \right\} \end{array}}{\widehat{\sigma} \vdash p, \underline{\star p} \overset{?}{\coloneqq} \widehat{vs}, [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} \widehat{\varrho_R}'''}$$

B Abstract Semantic Rules

Figure 12 shows the formal rules for executing the bottom-up visit-expression; we have omitted the collecting rules and some error handling rules to avoid presenting unnecessary details. We will further discuss the ideas behind the rules in a high-level fashion.

Executing visitors The evaluation rule for the visit-expression itself is mainly concerned with evaluating the target expression e to be traversed to a value, and then using a separate traversal relation to rewrite the value recursively with the sequence of cases $\underline{cs}$. The main item to notice is how it uses the value refined by the case patterns in case of failure (AE-VT-F), turning the result into successful execution (like in our running example in Sect. 2).

$$\sigma \vdash p \overset{?}{\coloneqq} v \underset{\text{match}}{\Longrightarrow} \underline{\rho} \qquad \widehat{\sigma} \vdash p \overset{?}{\coloneqq} \widehat{vs} \underset{\text{a-match}}{\Longrightarrow} \widehat{\varrho}$$

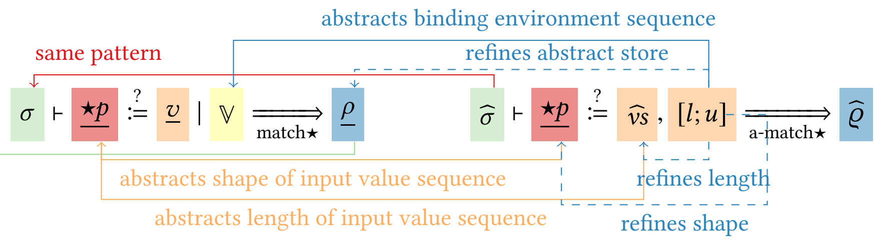

Figure 9. Relating abstract operational semantics (left) to the concrete operational semantics (right).

Expressions (General)

$$\text{AE-A}\ \frac{\{\!|\, x = e; \widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} \widehat{Res} \,|\!\}}{x = e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}} \qquad \text{AE-Sq}\ \frac{\{\!|\, e_1; e_2; \widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} \widehat{Res} \,|\!\}}{e_1; e_2; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}} \qquad \text{AE-C}\ \frac{\{\!|\, k(\underline{e}); \widehat{\sigma} \xRightarrow[\text{a-expr-cons}]{} \widehat{Res} \,|\!\}}{k(\underline{e}); \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\text{AE-St}\ \frac{\{\!|\, \{\underline{e}\}; \widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} \widehat{Res} \,|\!\}}{\{\underline{e}\}; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}} \qquad \text{AE-Fl}\ \frac{}{\text{fail}; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} [\text{fail} \mapsto (\cdot, \widehat{\sigma})]} \qquad \text{AES}\ \frac{\{\!|\, \underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \widehat{Res} \,|\!\}}{\underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res}}$$

Assignment Expression

$$\text{AE-A-S}\ \frac{\text{local } t\ x \lor \text{global } t\ x \qquad e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\text{success}, (\widehat{vs}, \widehat{\sigma}')) \in \widehat{Res} \qquad \widehat{vs} \mathrel{\widehat{:}} t' \qquad t' \mathrel{\widehat{<:}} t}{x = e; \widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} [\text{success} \mapsto (\widehat{vs}, \widehat{\sigma}'[x \mapsto (\text{ff}, \widehat{vs})])]}$$

$$\text{AE-A-Er}\ \frac{\text{local } t\ x \lor \text{global } t\ x \qquad e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\text{success}, (\widehat{vs}, \widehat{\sigma}')) \in \widehat{Res} \qquad \widehat{vs} \mathrel{\widehat{:}} t' \qquad t' \mathrel{\widehat{\not<:}} t}{x = e; \widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} [\text{error} \mapsto (\cdot, \widehat{\sigma}')]} \qquad \text{AE-A-Ex}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res}}{x = e; \widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Sequencing Expression

$$\text{AE-Sq-S}\ \frac{e_1, e_2; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\text{success}, ((\widehat{vs}_1, \widehat{vs}_2), \widehat{\sigma}')) \in \widehat{Res\star}}{e_1; e_2; \widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} [\text{success} \mapsto (\widehat{vs}_2, \widehat{\sigma}')]} \qquad \text{AE-Sq-Ex}\ \frac{e_1, e_2; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res\star}}{e_1; e_2; \widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Constructor Expression

$$\text{AE-C-S}\ \frac{\text{data } at = \ldots |k(\underline{t})| \ldots \qquad \underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\text{success}, (\underline{\widehat{vs}}, \widehat{\sigma}')) \in \widehat{Res\star} \qquad \underline{\widehat{vs}} \mathrel{\widehat{:}} \underline{t'} \qquad \underline{t'} \mathrel{\widehat{<:}} \underline{t}}{k(\underline{e}); \widehat{\sigma} \xRightarrow[\text{a-expr}]{} [\text{success} \mapsto (k(\underline{\widehat{vs}}), \widehat{\sigma}')]}$$

$$\text{AE-C-Er}\ \frac{\text{data } at = \ldots |k(\underline{t})| \ldots \qquad \underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\text{success}, (\underline{\widehat{vs}}, \widehat{\sigma}')) \in \widehat{Res\star} \qquad \underline{\widehat{vs}} \mathrel{\widehat{:}} \underline{t'} \qquad \exists i. t'_i \mathrel{\widehat{\not<:}} t_i}{k(\underline{e}); \widehat{\sigma} \xRightarrow[\text{a-expr}]{} [\text{error} \mapsto (\cdot, \widehat{\sigma}')]} \qquad \text{AE-C-Ex}\ \frac{\underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res\star}}{k(\underline{e}); \widehat{\sigma} \xRightarrow[\text{a-expr}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Set Expression

$$\text{AE-St-S}\ \frac{\underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\text{success}, (\underline{vs}, \widehat{\sigma}')) \in \widehat{Res\star}}{\{\underline{e}\}; \widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} [\text{success} \mapsto (\{\bigsqcup_i \widehat{vs}_i\}_{[0;|\underline{vs}|]}, \widehat{\sigma}')]} \qquad \text{AE-St-Ex}\ \frac{\underline{e}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res\star}}{\{\underline{e}\}; \widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Expression Sequences

$$\text{AES-Em}\ \frac{}{\varepsilon; \widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} [\text{success} \mapsto (\varepsilon, \widehat{\sigma})]} \qquad \text{AES-Mr}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\text{success}, (\widehat{vs}, \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{e'}; \widehat{\sigma}'' \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star}' \qquad (\text{success}, (\underline{\widehat{vs}'}, \widehat{\sigma}')) \in \widehat{Res\star}'}{e, \underline{e'}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} [\text{success} \mapsto ((\widehat{vs}, \underline{\widehat{vs}'}), \widehat{\sigma}')]}$$

$$\text{AES-Ex}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res}}{e, \underline{e'}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]} \qquad \text{AES-Ex}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\text{success}, (\widehat{vs}, \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{e'}; \widehat{\sigma}'' \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star}' \qquad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res\star}'}{e, \underline{e'}; \widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Figure 10. Abstract Operational Semantics Rules for Basic Expressions

Visit Expression

$$\text{AE-VT-S}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (\mathsf{success}, (\widehat{vs}, \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' \quad (\mathsf{success}, (\widehat{vs}', \widehat{\sigma}')) \in \widehat{Res}'}{\mathsf{bu\ visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} [\mathsf{success} \mapsto (\widehat{vs}', \widehat{\sigma}')]}$$

$$\text{AE-VT-F}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (\mathsf{success}, (\widehat{vs}, \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' \quad (\mathsf{fail}, (\widehat{vs}', \widehat{\sigma}')) \in \widehat{Res}'}{\mathsf{bu\ visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} [\mathsf{success} \mapsto (\widehat{vs}', \widehat{\sigma}')]}$$

$$\text{AE-VT-Ex1}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (exres, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res}}{\mathsf{bu\ visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} [exres \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

$$\text{AE-VT-Ex2}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (\mathsf{success}, (\widehat{vs}, \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' \quad (\mathsf{error}, (\widehat{resv}, \widehat{\sigma}')) \in \widehat{Res}'}{\mathsf{bu\ visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} [\mathsf{error} \mapsto (\widehat{resv}, \widehat{\sigma}')]}$$

Bottom-up Traversal of Single Value

$$\text{ABU-S}\ \frac{(\widehat{vs}'', \widehat{cvs}) \in \widehat{\mathsf{children}}(\widehat{vs}) \quad \underline{cs}; \widehat{cvs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \quad (\mathsf{success}, (\widehat{cvs}', \widehat{\sigma}')) \in \widehat{Res\star} \qquad \widehat{\mathsf{recons}}\ \widehat{vs}''\ \mathsf{using}\ \widehat{cvs}'\ \mathsf{to}\ \widehat{RCRes} \quad (\mathsf{success}, \widehat{vs}') \in \widehat{RCRes} \quad \underline{cs}; \widehat{vs}'; \widehat{\sigma}' \xRightarrow[\text{a-cases}]{} \widehat{Res}'}{\underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \widehat{Res}'}$$

$$\text{ABU-F}\ \frac{(\widehat{vs}'', \widehat{cvs}) \in \widehat{\mathsf{children}}(\widehat{vs}) \quad \underline{cs}; \widehat{cvs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \qquad (\mathsf{fail}, (\widehat{cvs}', \widehat{\sigma}')) \in \widehat{Res\star} \quad \widehat{\mathsf{recons}}\ \widehat{vs}''\ \mathsf{using}\ \widehat{cvs}'\ \mathsf{to}\ [\mathsf{success} \mapsto \widehat{vs}'] \quad \underline{cs}; \widehat{vs}'; \widehat{\sigma}' \xRightarrow[\text{a-cases}]{} \widehat{Res}'}{\underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \widehat{Res}'}$$

Bottom-up Traversal of Children

$$\text{ABUC-E}\ \frac{}{\underline{cs}; \varepsilon; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star\text{-go}]{} [\mathsf{fail} \mapsto (\varepsilon, \widehat{\sigma})]}$$

$$\text{ABUC-M}\ \frac{\underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}]{} \widehat{Res} \quad (\widehat{vfres}, (\widehat{vs}'', \widehat{\sigma}'')) \in \widehat{Res} \qquad \underline{cs}; \underline{\widehat{vs}'}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star}' \quad (\widehat{vfres}', (\underline{\widehat{vs}'''}, \widehat{\sigma}')) \in \widehat{Res\star}' \qquad \widehat{Res}'' = \widehat{\mathsf{vcombine}}(\widehat{vfres}, \widehat{vs}'', \widehat{vfres}', \underline{\widehat{vs}'''}, \widehat{\sigma}')}{\underline{cs}; \widehat{vs}, \underline{\widehat{vs}'}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star\text{-go}]{} \widehat{Res}''}$$

$$\text{ABUS-E}\ \frac{}{\underline{cs}; (\widehat{vs}, [0; u]); \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star\text{-go}]{} [\mathsf{fail} \mapsto ((\bot, 0), \widehat{\sigma})]}$$

$$\text{ABUS-M}\ \frac{u > 0 \quad \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}]{} \widehat{Res} \quad (\widehat{vfres}, (\widehat{vs}'', \widehat{\sigma}'')) \in \widehat{Res} \quad \underline{cs}; (\widehat{vs}, [l-1; u-1]); \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star}' \qquad (\widehat{vfres}', ((\widehat{vs}''', [l'; u']), \widehat{\sigma}')) \in \widehat{Res\star}' \quad \widehat{Res}'' = \widehat{\mathsf{vcombine}}(\widehat{vfres}, \widehat{vs}'', \widehat{vfres}', (\widehat{vs}''', [l'; u']), \widehat{\sigma}')}{\underline{cs}; (\widehat{vs}, [l; u]); \widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star\text{-go}]{} \widehat{Res}''}$$

Case Sequence

$$\text{ACS-E}\ \frac{}{\varepsilon; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-cases-go}]{} [\mathsf{fail} \mapsto (\widehat{vs}, \widehat{\sigma})]}$$

$$\text{ACS-M-O}\ \frac{\widehat{\sigma} \vdash p \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho} \quad (\widehat{vs}', \widehat{\sigma}', \widehat{\rho}^{?}) \in \widehat{\varrho} \qquad \widehat{\rho}^{?}; e; \widehat{\sigma}' \xRightarrow[\text{a-case}]{} \widehat{Res} \quad (rest, (\widehat{resv}, \widehat{\sigma}'')) \in \widehat{Res} \quad rest \neq \mathsf{fail}}{\mathsf{case}\ p \Rightarrow e, \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-cases-go}]{} [rest \mapsto (\widehat{resv}, \widehat{\sigma}'')]}$$

$$\text{ACS-M-F}\ \frac{\widehat{\sigma} \vdash p \overset{?}{\coloneqq} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho} \quad (\widehat{vs}', \widehat{\sigma}', \widehat{\rho}^{?}) \in \widehat{\varrho} \qquad \widehat{\rho}^{?}; e; \widehat{\sigma}' \xRightarrow[\text{a-case}]{} \widehat{Res} \quad (\mathsf{fail}, (\widehat{resv}, \widehat{\sigma}'')) \in \widehat{Res} \quad \underline{cs}; \widehat{vs}'; \widehat{\sigma}' \xRightarrow[\text{a-cases-go}]{} \widehat{Res}'}{\mathsf{case}\ p \Rightarrow e, \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-cases-go}]{} \widehat{Res}'}$$

Case

$$\text{AC-E}\ \frac{}{\bot; e; \widehat{\sigma} \xRightarrow[\text{a-case-go}]{} [\mathsf{fail} \mapsto (\cdot, \widehat{\sigma})]}$$

$$\text{AC-M-O}\ \frac{\widehat{\sigma}\ \widehat{\rho}; e \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (rest, (\widehat{resv}, \widehat{\sigma}'')) \in \widehat{Res} \quad rest \neq \mathsf{fail}}{\widehat{\rho}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-case-go}]{} [rest \mapsto (\widehat{resv}, \widehat{\sigma}'')]}$$

$$\text{AC-M-F}\ \frac{\widehat{\sigma}\ \widehat{\rho}; e \xRightarrow[\text{a-expr}]{} \widehat{Res} \quad (\mathsf{fail}, (\widehat{resv}, \widehat{\sigma}'')) \in \widehat{Res}}{\widehat{\rho}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-case-go}]{} [\mathsf{fail} \mapsto (\widehat{resv}, \widehat{\sigma})]}$$

Figure 11. Selected Abstract Operational Semantics Rules for Traversal