Verification of High-Level Transformations with Inductive Refinement Types

Ahmad Salim Al-Sibahi
DIKU/Skanned.com/ITU
Denmark

Thomas P. Jensen
INRIA Rennes
France

Aleksandar S. Dimovski
ITU/Mother Teresa University, Skopje
Denmark/Macedonia

Andrzej Wąsowski
ITU
Denmark

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 • **Theory of computation → Program verification**; **Program analysis**; **Abstraction**; *Functional constructs*; *Program schemes*; *Operational semantics*; Control primitives; • **Software and its engineering → Translator writing systems and compiler generators**; Semantics;

Keywords transformation languages, abstract interpretation, static analysis

ACM Reference Format:
Ahmad Salim Al-Sibahi, Thomas P. Jensen, Aleksandar S. Dimovski, and Andrzej Wąsowski. 2018. Verification of High-Level Transformations with Inductive Refinement Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3278122.3278125

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278125

1 Introduction

Transformations play a central role in software development. 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 [28], Stratego/XT [12], TXL [16], Uniplate [31] for Haskell, and Kiama [43] 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 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* (`visit`) to traverse and rewrite all statement lists containing a block to a flat list of statements. Visitors support various strategies, like the `bottom-up` strategy that traverses the abstract syntax tree starting from leaves toward the root.

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

- An *expressive pattern matching* language is used to nondeterministically find blocks inside a list of statements. The starred variable patterns `*xs` and `*zs` match arbitrary number of elements in the list, respectively before and after the `block(ys)` element. Rascal supports non-linear matching, negative matching and specifying patterns that match deeply nested values.
- The solve-loop (`solve`) performing the rewrite until a fixed point is reached (the value of s stops changing).

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 our PHP example, this would include:

- The transformation preserves the constructors used in the input: does not add or remove new types of PHP statements.
- The transformation produces flat statement lists, i.e., lists that 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 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. Schmidt-style abstract *operational* semantics [40] 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.

```
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

We proceed by presenting a running example in Sect. 2. We introduce the key constructs of Rascal in Sect. 3. Section 4 describes the modular construction of abstract domains. Sections 5 to 8 describe abstract semantics. 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 program of 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 one shall model interaction between these different types while maintaining precision.
- The traversal of syntax trees depends heavily on the *type and shape of input*, on a *complex program state*, and involves *unbounded recursion*. This challenges the inference of approximate invariants in a procedure that both terminates and provides useful results.
- Backtracking and exceptions in large programs introduce the possibility of *state-dependent non-local jumps*. This makes it difficult to statically calculate the control flow of target programs and have a compositional denotational semantics, instead of an operational one.

Figure 2 presents a small pedagogical example using visitors. The program performs expression simplification by traversing a syntax tree bottom-up and reducing multiplications by constant zero. We now survey the analysis techniques contributed in this paper, explaining them using this 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:

success cst (Nat) ≀ var (str) ≀ mult (Expr′, Expr′)

fail cst (Nat) ≀ var (str) ≀ mult (Expr′, Expr′)

where Expr′ = cst (suc (Nat)) ≀ var (str) ≀ mult (Expr′, Expr′).

We briefly interpret how to read this type. The bar ≀ denotes a choice between alternative constructors. If the input was rewritten during traversal (success, the first line) then the resulting syntax tree contains no multiplications by zero. All multiplications may only involve Expr′, which disallows the zero constant at the top level. Observe how the last alternative mult (Expr′, Expr′) contains only expressions of type Expr′, which in turn only allows multiplications by constants constructed using suc (Nat) (that is ≥ 1). If the traversal failed to match (fail, the second line), then the input did not contain any multiplication by zero to begin with and so does not the output, which has not been rewritten.

The success and failure happen to be the same for our example, but this is not necessarily always the case. Keeping separate result values allows retaining precision throughout the traversal, better reflecting concrete execution paths. 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) when it starts with shape mult (cst (Nat) , cst (Nat)). Since there is only one constructor, it will initially *recurse* down to traverse the contained values (children) creating a new recursion node (yellow, light shaded) 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 (blue, darker). 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). Here, we observe a problem: 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 idea is to detect the paths where execution recursively meets similar

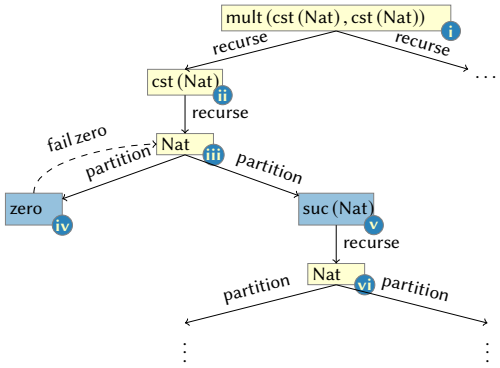

Figure 3. Naively abstractly interpreting the simplification example from Fig. 2 with initial input mult (cst (Nat) , cst (Nat)). The procedure does not terminate because of infinite recursion on Nat.

input, merging the new recursive node with the similar previous one, thus creating a loop in the execution tree [38, 40]. This loop is then resolved by a fixed-point iteration.

In Rabit, we propose *partition-driven trace memoization*, which works with potentially unbounded input like the inductive type refinements that are supported by our abstraction. We detect cycles by maintaining 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 map:

- If the last processed refinement type representing the input i' 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 input refinement types to a new value $i'' = i' \nabla i$, update the memoization map and continue execution with i''. The 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. The memoization map is updated to map the general type of i'' (not refined, for instance Nat) to map to a pair (i'', o), where the first component denotes the new input i'' refinement type and the second component denotes the corresponding output o refinement type; initially, o is set to $\bot$ and then changed to the result of executing input i'' repeatedly until a fixed-point is reached.

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 show the memoization map inside a framed orange 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 $\bot$. 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 ($\bot$ initially).

Trace Partitioning We *partition* [36] the abstract value Nat along its constructors: zero and suc $(\cdot)$ (Fig. 4). This partitioning is key to maintain precision during the abstract interpretation. As in Fig. 3, the left branch fails immediately, since no pattern in Fig. 2 matches zero. The right branch descends into a new recursion over Nat, with an updated memoization table. This run terminates, due to a hit in the memoization map, returning $\bot$. After returning, the value of suc (Nat) should be reconstructed with the result of traversing the child Nat, but since the result is $\bot$ there is no value to reconstruct with, so $\bot$ 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: the child Nat is replaced by zero and fail suc (zero) is returned (dashed arrow from 3 to 1). 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 $o_{\text{prev}} = \text{Nat}$ 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. When inductive shapes (e.g., Expr) refer to other inductive shapes (e.g., Nat), it is necessary to run 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 with 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:

$$\begin{aligned}&\text{success cst}\,(\text{Nat})\\&\text{fail mult}\,(\text{cst}\,(\text{suc}\,(\text{Nat}))\,,\ \text{cst}\,(\text{suc}\,(\text{Nat})))\end{aligned}$$

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 [28], relying on the concrete semantics from Rascal Light [2].

We consider algebraic data types (at) and finite sets ($\text{set}\langle t\rangle$) 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.

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

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:

$$\begin{aligned}e &::= x \in \text{Var} \mid x = e \mid e; e \mid k(\underline{e}) \mid \{\underline{e}\} \mid \text{fail} \mid \text{visit}\ e\ \underline{cs}\\ cs &::= \text{case}\ p \Rightarrow e\end{aligned}$$

Visitors are a key construct in Rascal. A visitor visit e $\underline{cs}$ traverses recursively the 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 bottom-up visitors, but Rabit (Sect. 9) supports all strategies of Rascal.

Notation We write $(x, y) \in f$ to denote the pair (x, y) such that $x \in \text{dom}\ f$ 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 contracted using an underlining $\underline{e}$. The empty sequence is written 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

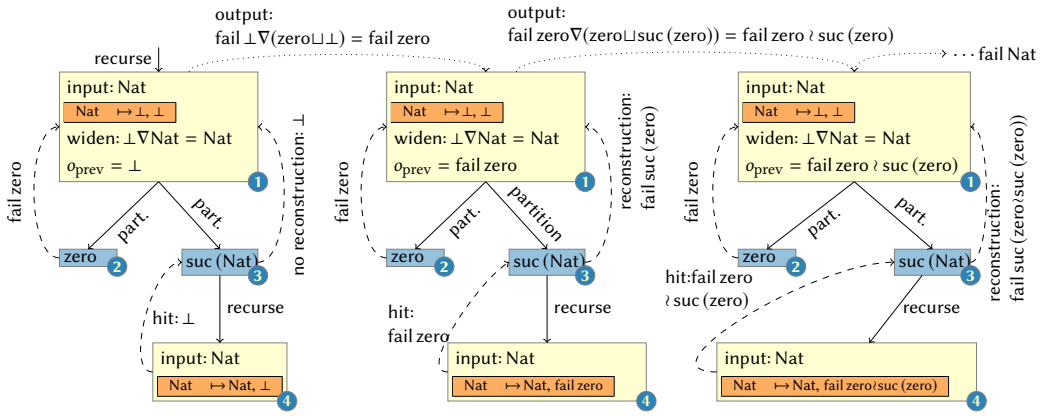

Figure 4. Three iterations of a fixed point computation for input Nat. Iterations are separated by dotted arrows on top

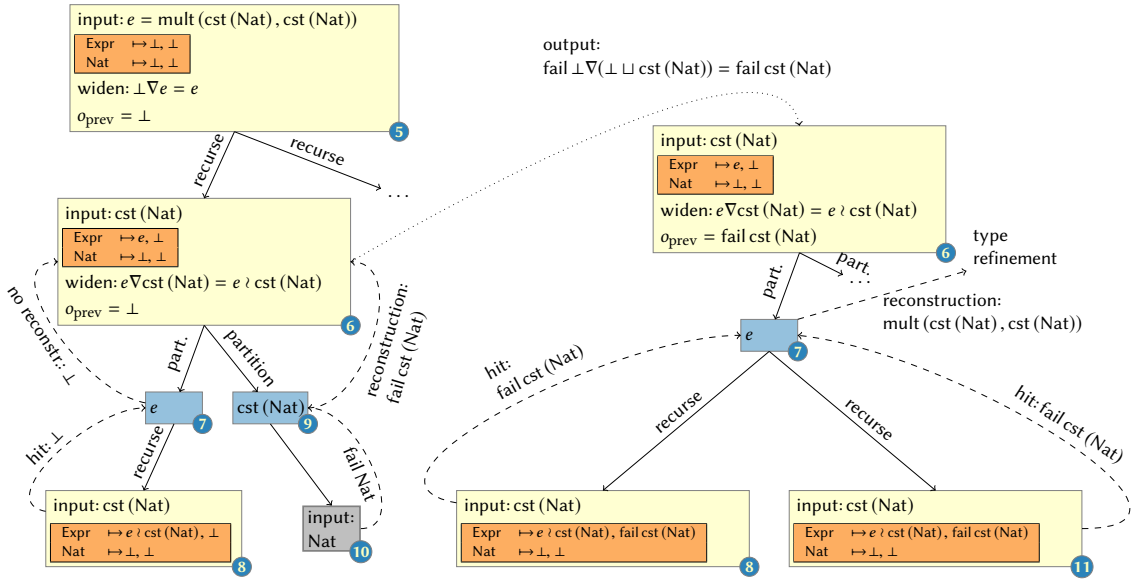

Figure 5. A prefix of the abstract interpreter run for $e = \text{mult}(\text{cst}(\text{Nat}), \text{cst}(\text{Nat}))$. Fragments of two iterations involving node 6 are shown, separated by a dotted arrow.

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 for modular composition. Modularity is key for transformation languages, which manipulate a large variety of values. The design allows easily replacing abstract domains for particular types of values, as well as adding support for new value types. We want to construct an abstract value domain $\widehat{vs} \in \widehat{\text{ValueShape}}$ 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 [8, 15, 41]. The idea is to define domains parametrically—i.e. in the form $\widehat{\text{F}}(\underline{\widehat{\text{E}}})$—so that abstract domains for subcomponents are taken as parameters, and explicit recursion is handled separately. We use standard domain combinators [48] to combine individual domains into the abstract value domain.

Set Shape Domain. Let Set(E) denote the domain of sets consisting of elements taken from E. We define abstract finite sets using abstract elements $\{\widehat{e}\}_{[l;u]}$ from a parameterized domain $\widehat{\text{SetShape}}(\widehat{\text{E}})$. The component from the parameter domain ($\widehat{e} \in \widehat{\text{E}}$) represents the abstraction of the shape of elements, and a non-negative interval component $[l;u] \in \widehat{\text{Interval}}^+$ is used to abstract over the cardinality (so $l, u \in \mathbb{R}^+$

and $l \leq u$). The abstract set element acts as a reduced product between $\widehat{e}$ and $[l;u]$ and the lattice operations follow directly.

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

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

Example 4.1. Let $\widehat{\mathsf{Interval}}$ be a domain of intervals of integers (a standard abstraction over integers). We can concretize abstract elements from $\widehat{\mathsf{SetShape}}(\widehat{\mathsf{Interval}})$ to a set of possible sets of integers from $\wp(\mathsf{Set}(\mathbb{Z}))$ 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 [24, 39, 50] 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 $\gamma_{\widehat{\mathsf{E}}} \in \widehat{\mathsf{E}} \to \wp(\mathsf{E})$, we can create a concretization function for the data shape domain

$$\gamma_{\widehat{\mathsf{DS}}} \in \widehat{\mathsf{DataShape}}(\widehat{\mathsf{E}}) \to \wp(\mathsf{Data}(\mathsf{E}))$$

where $\mathsf{Data}(\mathsf{E}) = \left\{ k(\underline{v}) \;\middle|\; \exists \text{ a type } at.\ k(\underline{v}) \in [\![at]\!] \wedge \underline{v \in \mathsf{E}} \right\}$. 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. 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:

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

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 $X = \mathsf{F}(X)$ [41, 44, 48], 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:

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

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.

4.1 Abstract State Domains

We now explain how to construct abstractions of states and results when executing Rascal programs.

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:

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

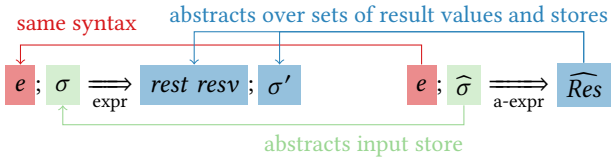

Figure 6. Relating concrete semantics (left) to abstract semantics (right).

Abstract Result Domain. Traditionally, abstract control flow is handled using a collecting denotational semantics with continuations, or by explicitly constructing a control flow graph. These methods are non-trivial to apply for a rich language like Rascal, especially considering backtracking, exceptions and data-dependent control flow introduced by visitors. A nice side-effect of Schmidt-style abstract interpretation is that it allows abstracting control flow directly.

We model different type of results—successes, pattern match failures, errors directly in a $\widehat{\text{ResSet}}$ domain which keeps track of possible results with each its own separate store. Keeping separate stores is important to maintain precision around different paths:

$$rest \in \widehat{\text{ResType}} ::= \text{success} \mid exres$$

$$exres ::= \text{fail} \mid \text{error} \quad \widehat{resv} \in \widehat{\text{ResVal}} ::= \cdot \mid \widehat{vs}$$

$$\widehat{Res} \in \widehat{\text{ResSet}} = \widehat{\text{ResType}} \rightharpoonup \widehat{\text{ResVal}} \times \widehat{\text{Store}}$$

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

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

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 [10, 40]. The creative work is thus 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 produce a terminating static analysis that approximates an infinite number of concrete traces.

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:

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

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:

$$\text{E-V-S}\ \frac{x \in \text{dom}\ \sigma}{x;\sigma \underset{\text{expr}}{\Longrightarrow} \text{success}\ \sigma(x);\sigma} \qquad \text{E-V-Er}\ \frac{x \notin \text{dom}\ \sigma}{x;\sigma \underset{\text{expr}}{\Longrightarrow} \text{error};\sigma}$$

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. $x \in \text{dom}\ \sigma$ with $\widehat{\sigma}(x) = (b, \widehat{vs})$ and a check on b. We obtain two execution rules:

$$\text{AE-V-S}\ \frac{\widehat{\sigma}(x) = (b, \widehat{vs})}{x;\widehat{\sigma} \underset{\text{a-expr-v}}{\Longrightarrow} [\text{success} \mapsto (\widehat{vs}, \widehat{\sigma})]} \qquad \text{AE-V-ER}\ \frac{\widehat{\sigma}(x) = (\text{tt}, \widehat{vs})}{x;\widehat{\sigma} \underset{\text{a-expr-v}}{\Longrightarrow} [\text{error} \mapsto (\cdot, \widehat{\sigma})]}$$

   Observe when b is true, both a success and failure may occur, and we need rules to cover both cases.
3. Create a rule that collects all possible evaluations of the syntax-specific judgment rules, e.g. AE-V for variables:

$$\text{AE-V}\ \frac{\{\!|x;\widehat{\sigma} \underset{\text{a-expr-v}}{\Longrightarrow} \widehat{Res}'|\!\}}{x;\widehat{\sigma} \underset{\text{a-expr}}{\Longrightarrow} \widehat{Res}'}$$

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

It is possible to translate the operational semantics rules for other basic expressions using the presented steps [4, Appendix B]). 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 $e; \widehat{\sigma} \overset{\text{a-expr}}{\Longrightarrow} \widehat{Res}$ and $(\text{success}, (\widehat{vs}, \widehat{\sigma}')) \in \widehat{Res}$, as compared to $e; \sigma \overset{\text{expr}}{\Longrightarrow} \text{success } v; \sigma'$ in the concrete semantics.
- Typing is now done using abstract judgments $\widehat{vs} \mathbin{\widehat{:}} t$ and $t \mathbin{\widehat{<:}} t'$. In particular, type t is an abstract subtype of type t' ($t \mathbin{\widehat{<:}} t'$) if there is a subtype t'' of t ($t'' <: t$) that is also a subtype of t' ($t'' <: t'$). This implies that $t \mathbin{\widehat{<:}} t'$ and $t \mathbin{\widehat{\not<:}} t'$ are non-exclusive.
- To check whether a particular constructor is possible, we use the abstract function $\widehat{\text{unfold}}(\widehat{vs}, t)$ that produces a refined value of type t if possible—splitting alternative constructors—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. 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 $\star p$; each sub-pattern in the sequence 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 variables, 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*. The key aspects of Rabit in handling pattern matching is how we maintain precision by *refining* input values on pattern matching successes and failures.

6.1 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:

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

A value v matches a pattern p ($v \models p$) iff 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 Fig. 7a.

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. We revisit the running example to understand how the data type values are matched. We consider matching the following 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:

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

Similarly, 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 ($\widehat{vs} \mathbin{\widehat{\models}} p$) if there exists a more

$k(\underline{v}) \models_\rho k(\underline{p})$	**iff**	$\underline{t}$ are parameter types of k **and** $\underline{v} : \underline{t'}$ **and** $\underline{t' <: t}$ **and** $\underline{v} \models^\star_\rho \underline{p}$
$v \models_\rho x$	**iff**	$\rho(x) = v$
$\{\underline{v}\} \models_\rho \{\underline{\star p}\}$	**iff**	$\underline{v} \models^\star_\rho \underline{\star p}$
$\varepsilon \models^\star_\rho \varepsilon$	**always**	
$v, \underline{v'} \models^\star_\rho p, \underline{\star p'}$	**iff**	$v \models_\rho p$ **and** $\underline{v'} \models^\star_\rho \underline{\star p'}$
$\underline{v}, \underline{v'} \models^\star_\rho \star x, \underline{\star p'}$	**iff**	$\rho(x) = \{\underline{v}\}$ **and** $\underline{v'} \models^\star_\rho \underline{\star p'}$

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

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

(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

precise value $\widehat{vs}'$ (so $\widehat{vs}' \sqsubseteq \widehat{vs}$) and an abstract binding environment $\widehat\rho$ with $\mathbf{dom}\ \widehat\rho = \mathrm{vars}(p)$ 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.

6.2 Computing Pattern Matches

The declarative satisfiability semantics of patterns, albeit clean, is not directly computable. In Rabit, we rely on an abstract operational semantics [4, Appendix A] using the technique presented in Sect. 5. The interesting ideas are in the refining semantic operators that we now discuss.

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 $\widehat{vs} \mathrel{\widehat{=}} \widehat{vs}'$. 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:

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

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 exclude k from the refined value. For a data type abstraction exclusion removes the pattern constructor from the possible choices

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

and does not change the input shape otherwise.

7 Traversals

First-class traversals are a key feature of high-level transformation languages, since they enable effectively transforming large abstract syntax trees. We will focus on the challenges for bottom-up traversals, but they are shared amongst all strategies supported in Rascal. 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 a new value using the rewritten children and finally traversing the reconstructed value. The main challenge is

handling traversal of children, whose representation and thus execution rules depend on the particular abstract value.

Concretely, the $\widehat{\text{children}}(\widehat{vs})$ function returns a set of pairs $(\widehat{vs}', \widehat{cvs})$ 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 the representation of children is a pair $(\underline{\widehat{vs}''}, [l; u])$ with the first component representing the shape of elements and the second representing their count. For example,

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

and $\widehat{\text{children}}(\{\text{Expr}\}_{[1;10]}) = \{(\{\text{Expr}\}_{[1;10]}, (\text{Expr}, [1; 10]))\}$. Note how the $\widehat{\text{children}}$ function maintains precision by partitioning the alternatives for data-types, when traversing each corresponding sequence of value shapes for the children.

Traversing Children. The shape of execution rules depend on the representation of children; this is consistent with the requirements imposed by Schmidt [40]. For heterogeneous sequences of value shapes $\underline{\widehat{vs}}$, the execution rules iterate through the sequence recursively traversing each element. 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. 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. 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.

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, the main technique to handle recursion is *trace memoization* [38, 40]. 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 itself, like a while-loop or traversal.

The main challenge when recursing over inputs from infinite domains, is to determine *when* to merge recursive paths together to correctly over-approximate concrete executions. We present an extension that is still terminating, sound and, additionally, 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 $\widehat{M} \in \widehat{\text{PInput}} \rightharpoonup \widehat{\text{Input}} \times \widehat{\text{Output}}$ 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.

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; therefore our function is continuous and a fixed-point exists.

[2] Provided that we bound the depth of type parameters of collections.

9 Experimental Evaluation

We demonstrate the ability of Rabit to verify inductive shape properties, using five transformation programs, where three are classical and two are extracted from open source projects.

Negation Normal Form (**NNF**) transformation [26, 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 [7, 49], translates for-loops and switch-statements to while-loops and nested if-statements, respectively.

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 [42] 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), type inference, an interpreter and a compiler.

P10 Simplification procedure produces an expression with no additions with 0, multiplications with 1 or 0, subtractions with 0, logical expressions with constant operands, and if-expressions with constant 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 subset 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 can be hard to generalize the results. We mitigated this by selecting transformations that are realistic and vary in authors and purpose. While translating the programs to Rascal Light, we strived to minimize the amount of changes, but bias cannot be ruled out entirely.

Implementation. We have implemented the abstract interpreter in a prototype tool, Rabit[4], 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. Fine-tuning memoization strategies to the different looping constructs and recursive calls

By default, we use the top element $\top$ specified as input, but the user can specify the initial data-type refinements, store and parameters, to get more precise results. 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 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.

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 Rabit, and whether the properties have been verified. Since we verify the results on the abstract shapes, the programs are shown to be correct for all possible concrete inputs satisfying the given properties. We remark that all programs use the high-level expressive features of Rascal and are succinct compared to general purpose code.

The runtime, varying from single seconds to less than a minute, is reasonable. All, but two, properties were successfully verified. The reason that our tool runs slower on the DSO transformation is that it contains function calls and we rely on inlining for interprocedural analysis.

Lines 1–2 in Fig. 8 show the input refinement type FIn for the normalization procedure. The inferred inductive output

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

[4] https://github.com/itu-square/Rascal-Light

Figure 8. Initial and inferred refinement types for NNF

```
1 data FIn = and(FIn, FIn) | atom(str) | neg(FIn)
2            | imp(FIn, FIn) | or(FIn, FIn)
3
4 data FOut = and(FOut, FOut) | atom(str)
5             | neg(atom(str)) | or(FOut, FOut)
```

type FOut (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 formulation of negation normal form as an inductive data type.

10 Related Work

We start with discussing techniques that could make Rabit verify properties like P3 and P7. To verify P3, we need to be able to relate field names to their corresponding definitions in the class. Relational abstract interpreteration [32] allows specifying constraints that relate values across different variables, even inside and across substructures [14, 25, 30]. 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. A possible solution could be to allow *abstract attributes* that extract additional information about the abstracted structures [11, 34, 45]. A generalization of the multiset abstraction [33], could be useful to track e.g., the auxiliary statement count, and show that they decrease using multiset-ordering [23]. Other techniques [5, 14, 47] support inferring inductive relational properties for general data-types—e.g, binary tree property—but require a pre-specified structure to indicate where refinement can happen.

Cousot and Cousot [19] present a general framework for modularly constructing program analyses, but it requires languages with compositional control flow. Toubhans, Rival and Chang [37, 46] develop a modular domain design for pointer-manipulating programs, whereas our domain construction focuses on abstracting pure heterogeneous data-structures.

There are similarities between our work and verification techniques based on program transformation[22, 29]. Our systematic exploration of execution rules for abstraction is similar to *unfolding*, and widening is similar to *folding*. The main difference is that abstract interpretation widens at syntactic program points using rich domains, while folding happens dynamically on the semantic execution graph.

Definitional interpreters have been suggested as a technique for building compositional abstract interpreters [21]. They rely on a *caching* algorithm to ensure termination, similarly to ordinary finite input *trace memoization* [38]. Similarly, Van Horn and Might [27] present a systematic framework to abstract higher-order functional languages. They rely on store-allocated continuations to handle recursion, which is kept finite during abstraction to ensure a terminating analysis. We focused on providing a more precise widening based on the abstract input value, which was necessary for verifying the required properties in our evaluation.

Modern SMT solvers supports reasoning with inductive functions defined over algebraic data-types [35]. The properties they can verify are very expressive, but they do not scale to large programs like transformations. Possible constructor analysis [6] has been used to calculate the actual dependencies of a predicate and make flow-sensitive analyses more precise. This analysis works with complex data-types and arrays, but only captures the prefix of the target structures.

Techniques for model transformation verification on static analysis [20] have been suggested, but are on verification of types and undefinedness properties. Symbolic execution has previously been suggested [3] as a way to validate high-level transformation programs, but it targets test generation rather than verification of properties. Semantic typing [9, 13] has been used to infer recursive type and shape properties for language with high-level constructs for querying and iteration. However, they only consider small calculi compared to Rascal Light, and our evaluation is more extensive.

Table 1. Time and success rate for analyzing programs and properties presented earlier this section.

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	✓

11 Conclusion

Our goal was to use abstract interpretation to give a solid semantic foundation for analyzing programs in modern high-level transformation languages. We designed and implemented a Schmidt-style abstract interpreter, Rabit, including *partition-driven trace memoization* that supports infinite input domains. This worked well for Rascal, and can be adapted for similar languages with complex control flow. The modular construction of abstract domains was vital for handling a language of this scale and complexity. We evaluated Rabit on classical and open source transformations, by verifying a series of sophisticated shape properties for them.

Acknowledgments

We thank Paul Klint, Tijs van der Storm, Jurgen Vinju and Davy Landman for discussions on Rascal and its semantics. We thank Rasmus Møgelberg and Jan Midtgaard for discussions on correctness of the recursive shape abstractions. This material is based upon work supported by the Danish Council for Independent Research under Grant No. 0602-02327B and Innovation Fund Denmark under Grant No. 7039-00072B.

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] Ahmad Salim Al-Sibahi, Thomas P. Jensen, Aleksandar S. Dimovski, and Andrzej Wąsowski. 2018. Verification of High-Level Transformations with Inductive Refinement Types. *ArXiv e-prints* (Sept. 2018). arXiv:cs.PL/1809.06336 https://arxiv.org/abs/1809.06336

[5] 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

[6] 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

[7] 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

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

[9] 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

[10] 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

[11] 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

[12] 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

[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] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2014. Program verification via iterated specialization. *Sci. Comput. Program.* 95 (2014), 149–175. https://doi.org/10.1016/j.scico.2014.05.017

[23] 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

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

[25] 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

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

[27] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. https://doi.org/10.1145/1863543.1863553

[28] 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

[29] Alexei P. Lisitsa and Andrei P. Nemytykh. 2015. Finite Countermodel Based Verification for Program Transformation (A Case Study). In *Proceedings of the Third International Workshop on Verification and Program Transformation, VPT@ETAPS 2015, London, United Kingdom, 11th April 2015. (EPTCS)*, Alexei Lisitsa, Andrei P. Nemytykh, and Alberto Pettorossi (Eds.), Vol. 199. 15–32. https://doi.org/10.4204/EPTCS.199.2

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

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

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

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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

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

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

[50] 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