Foundational Analysis Techniques for High-Level Transformation Programs

Ahmad Salim Al-Sibahi

Advisors:
Andrzej Wąsowski
Aleksandar Dimovski
Submitted: September 2017

IT UNIVERSITY OF COPENHAGEN

ليس التّقدم بتحسين ما كان
بل بالسير نحو ما سيكون

جبران خليل جبران

Progress is beyond improving what is;
Progress is moving towards what will be.

Gibran Khalil Gibran

Abstract

High-level transformation languages like TXL and Rascal, simplify writing and maintaining transformations such as refactorings and optimizations, by providing specialized language constructs that include powerful pattern matching, generic traversals, and fixed-point iteration. This expressivity unfortunately makes it hard to reason about the semantics of these languages, and there exists few automated validation and verification tools for these languages.

The goal of this dissertation is to develop foundational techniques and prototype tools for verifying transformation programs, based on techniques from traditional programming language research. At first, I present a systematic analysis of declarative high-level transformation languages, like QVT and ATL, seeking to clarify their expressiveness. The analysis showed that virtually all examined languages were Turing-complete and thus could not be treated specially for verification.

I describe a joint effort in designing and validating an industrial modernization project using the high-level transformation language TXL. The transformation was validated using the translation validation and symbolic execution, catching 50 serious bug cases in an otherwise well-tested expert-written transformation, and thus showing the need for verification tools in this domain.

Using these experiences, I develop a formal symbolic execution algorithm for a small transformation language capturing key high-level constructs. A proto-type white-box test generation tool was implemented using the algorithm, and evaluated on a series of refactorings and model transformations comparing favorably against baseline black-box test generator in terms of test coverage.

I present the first formal operational semantics for a large subset of Rascal, called Rascal Light, which has been validated by formally proving a series of semantic properties regarding strong typing, safety and termination. This creates a solid basis for further foundational work on high-level transformation languages.

Finally, I develop a formal abstract interpreter for Rascal Light, with a modular abstract domain design and a Schmidt-style abstract operational semantics, adapting the idea of widening by *trace memoization* to work with inputs from infinite domains. The technique is evaluated on real Rascal transformations, showing that it is possible to effectively verify target type and shape properties.

Resumé

Højniveau transformationssprog såsom TXL og Rascal gør det nemmere at skrive og vedligeholde transformationer, f.eks. refaktoreringer og optimeringer, ved at understøtte specielle sprogskonstruktioner inklusiv kraftfulde mønstersamsvarsoperationer (orig. *pattern matching*), generisk gennemgang af datastrukturer, og fikspunktsiteration. Denne ekspresivitet gør det dog desværre svært at få en fuld forståelse for semantikken til disse sprog, og der eksisterer derfor kun få automatiske værktøjer der understøtter validering og verifikation af dem.

Hovedformålet med denne afhandling er at udvikle fundamentale teknikker og værktøjsprototyper som understøtter verifikation af transformationsprogrammer, baseret på teknikker fra traditionel programmeringssprogsforskning. I starten, præsenterer jeg en systematisk analyse af deklarative højniveau transformationssprog, såsom QVT og ATL, for at tydeliggøre deres ekspresivitet. Analysen viste at stort set alle undersøgte sprog var Turing-komplette og kunne derfor ikke blive særbehandlet i verifikationsøjemed.

Jeg beskriver en fælles indsats hvis formål er at designe og validere et industrielt moderniseringsprojekt ved brug af højniveau transformationssproget TXL. Transformationen var valideret ved brug af oversættelsesvalidering (orig. *translation validation*) og symbolsk eksekvering, og opdagede 50 seriøse programfejl i en ellers veltestet transformation skrevet af en ekspert, hvilket fremviser behovet for verifikationsværktøjer i dette område.

Gørende brug af disse erfaringer, udvikler jeg en formel symbolsk eksekveringsalgoritme for et lille transformationssprog der modellerer de vigtigste højniveau konstruktioner. Jeg implementerede en white-box testgenereringsværktøjsprototype ved brug af algoritmen, og evaluerede denne vha. en række refaktoreringer og modeltransformationer, hvilket viste gode resultater i forhold til programgrensdækning sammenlignet med black-box testgenereringsværktøjet som blev brug som basislinje.

Jeg præsenterer den første formelle operationelle semantik for en stor delmængde af sproget Rascal, kaldet Rascal Light, hvilket er valideret ved at bevise en række semantiske egenskaber omkring stært typedisciplin, sikkerhed, og programafslutning. Dette lægger et solidt fundament for videre formelt arbejde omkring højniveau transformationssprog.

Afslutningsvis, udvikler jeg en formel abstrakt fortolker til Rascal Light, med et tilhørende modulært abstrakt domæne og en abstrakt operationel semantik i stil med Schmidt, og tilpasser konceptet om udvidelse (orig. *widening*) via memoisering af programspor til at virke med input fra uendeligt store domæner. Teknikken evalueres ved brug af rigtige Rascal transformationsprogrammer, hvilket viser at det er muligt at effektivt verificere de egenskaber omkring typer og form som vi havde til mål.

Foreword

How long a time is three years? It is too long, you get to know your project so well that you become an expert in its extremely narrow subject. It is too short, it ended at the moment you just started to know enough in the field to dare to tackle more complex, more interesting problems. Maybe, it is just enough time to earn a PhD.

The thing I recall most is my failing attempts at solving the problems I encountered, moreso than the successes that I present in this dissertation. Yet, these failures, remaining hidden, under the carpet, in my mind, are what makes research all the worthwile. For how can we otherwise make worthwile contributions, if there is nothing to dare; is it truly research if it can not fail, if it does not fail, if it will not fail *again*?

Acknowledgements

I would like to thank my family, who helped me with material and moral support, and especially during my PhD studies. I would like to thank the Danish society for funding my education throughout my life, all the way up to this point. This project in particular was supported mainly by The Danish Council for Independent Research under a Sapere Aude project grant no. 0602-02327B., VARIETE, and partially by ARTEMIS JU under grant agreement n°295397 together with Danish Agency for Science, Technology and Innovation.

I would like to thank my supervisors, Andrzej Wąsowski and Aleksandar Dimovski for comments, discussions and guidance during my studies, and for putting up with my occasional cloud-headedness. I would like thanks the rest of my collaborators, Claus Brabrand, Thomas Jensen, Alexandru Florin Iosif-Lazăr, Juha Erik Savolainen and Krzysztof Sierszecki, for their contributions to this project as well. I would like to thank the comittee, Riko Jacob, Xavier Rival and Tijs van der Storm, for evaluating this dissertation and providing valuable feedback.

I would like to thank Thomas Jensen, again, and his group, Celtique, at INRIA Rennes, for the great welcome I received during my stay-abroad, and for the interesting discussions during the reading groups. I would like to thank Lars Birkedal and his group at Aarhus University, and similarly Constantin Enea and Michaela Sighireanu and their group at Paris Diderot University (VII) for giving me the opportunity to present my work and discuss it during the two short visits I had during my PhD.

I thank Karl Portratz for introducing our team to the industrial modernization project at Danfoss, Rolf-Helge Pfeiffer for implementing the modernization transformation and Claus Brabrand for discussions on its design. I would like to thank Paul Klint and the Rascal team at CWI Amsterdam for answering my questions about the formal Rascal semantics in details and providing corrections on the corresponding report. I would like to thank Rasmus Ejlers Møgelberg for discussions about the correctness of the fixed-point domain abstraction theorem in the Rascal abstract interpreter.

Formal Notation

- For a binary relation $\mathsf{r} \subseteq \mathrm{A} \times \mathrm{A}$, r^* is used to denote the reflexive-transitive closure, and similarly r^+ for the transitive (non-reflexive) closure.
- For a set A, $\wp(A)$ is used to denote the power set.
- For a function $f \in A \rightarrow B$, $\operatorname{dom} f$ is used to represent all defined inputs i.e., $\{x \mid \exists y.y = f(x)\}$, $\operatorname{img} f$ to represent all defined outputs $\{y \mid \exists x.y = f(x)\}$, and $\operatorname{graph} f$ to represent the set $\{(x, f(x)) \mid x \in \operatorname{dom} f\}$.
- For a function $f \in A \rightarrow B$, $\operatorname{dom} f$, $f[a \mapsto b]$ is used to represent function updates, so that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](a') = f(a')$ when $a' \neq a$.
- Approximate (symbolic or abstract) semantic components, sets and operations are marked with a wide hat $\widehat{a}$.
- A sequence of es is represented by either marked by an underline $\underline{e}$ or explicitly summarized $e_1, \ldots, e_\mathrm{n}$, the empty sequence is denoted by ε and the concatenation of sequences $\underline{e_1}$ and $\underline{e_2}$ is denoted $\underline{e_1}, \underline{e_2}$.
- Notation is overloaded in an intuitive manner for operations on sequences, so given a sequence $\underline{v}$, v_i denotes the ith element in the sequence, and $\underline{v : t}$ denotes the sequence $v_1 : t_1, \ldots, v_\mathrm{n} : t_\mathrm{n}$.
- Operators from different sets are distinguished with a subscript, e.g., $\top_\mathrm{A}$ represents the top element of domain A; the subscripts are left out when the intented use of operators is clear from context.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Domain-specific languages or *DSL*s [Bentley, 1986; Hudak, 1996; van Deursen et al., 2000; Mernik et al., 2005; Fowler, 2011] allow expressing models and programs pertient to a business domain (e.g., finance [Christerson et al., 2013], robotics [Nordmann et al., 2014], cryptography [Erkök and Matthews, 2009]) using a vocabulary familiar to domain users. These languages eschew generality in favor of providing a clear correspondence between domain requirements and models, being important for maintainability and evolution of modelled systems, and easing communication between domain users and technical experts.

DSL engineering relies heavily on the use of *transformations*, which allow converting between structured models or programs. Examples of such transformations include:

Model Translation Translating between structured models preserving consistency [Jouault et al., 2008], e.g., a class diagram to a relational diagram.

Desugaring Transformation of constructs from a language extended with *syntactic sugar* [Landin, 1964]—that primarily target easing programming—to constructs in a core language, e.g., translating **for**-loops to **while**-loops.

Refactoring Restructuring of source code [Fowler, 1999] to increase readability and maintainability while largely preserving intented semantics, e.g., renaming the field of a class.

Code Generation Converting constructs from one language to another for execution or further processing[Aho et al., 1986b], e.g., generating assembly code from an imperative language.

Optimization Transforming programs possibly given some static information in order to improve runtime and memory use [Aho et al., 1986c], e.g., software pipelining and partial evaluation [Jones et al., 1993].

Most practical DSLs have large abstract syntax—often spanning hundreds of syntactic elements—and correspondingly rich semantics; as a result, this makes writing transformations for these languages a tedious and error-prone process, more so using traditional programming languages like Java [Gosling et al., 2014], C$^\sharp$ [Hejlsberg et al., 2011] or Haskell [Marlow, 2010].

To simplify the task of writing and maintaining such transformations, specialized high-level transformation languages have been proposed across different communities in software language engineering (e.g. Rascal [Klint et al., 2011], Stratego/XT [Bravenboer et al., 2008]), programming languages (e.g. TXL [Cordy, 2006], Uniplate [Mitchell and Runciman, 2007]) and model-driven engineering (e.g. QVT [Object Management Group, 2016], ATL [Jouault et al., 2008], ETL [Kolovos et al., 2008]). Common for these languages is that they all provide rich features for traversing and manipulating abstract syntax, powerful pattern matching and querying, backtracking and generalized looping constructs.

Figure 1.1 shows an example Rascal programs which uses these rich features to inline constant declarations in a simple expression language. The `constsValid` function pattern matches on the declarations inside a module, checking whether there are duplicate declarations of constants using arbitrary elements match patterns `*_` to skip elements and non-linear patterns to check existence of two equal names (notice the two instances of `x` in the same pattern). The `inlineConsts` function iterates over all constant declarations using a pattern matching **for**-loop, removing the constant declaration from the list of declarations and traversing the body expression using **bottom-up visit** replacing all references to the constant with the provided value.

I will further illustrate how these features are concretely realized in Rascal:

```
data Decl = constdecl(str name, int val)
          | vardecl(str name);
data Expr = mult(Expr l, Expr r) | add(Expr l, Expr r)
          | var(str name) | const(int val);
data Mod = \mod(list[Decl] decls, Expr body);

bool constsValid(Mod m) {
  switch (m.decls) {
    case [*_, constdecl(x, _), *_,
               constdecl(x, _), *_]: return false;
    case _: return true;
  }
}
Mod inlineConsts(Mod m) {
  assert constsValid(m);
  for (cd: constdecl(x, v) <- m.decls) {
    m.decls = m.decls - cd;
    m.body = bottom-up visit(m.body) {
      case var(x) => const(v)
    };
  };
  return m;
}
```

Figure 1.1: Constant Inlining Transformation

- Traversals use **visit** which allows automated rewriting of abstract syntax trees using a sequence of pattern matching cases using either **bottom-up** ordering as in the example or **top-down** ordering.
- Pattern matching supports a rich set of operations, such as the constructor matching, arbitrary elements matching, and non-linear matching presented in the example. Other operations include type-based matching, negative matching, and deep matching that pattern matches arbitrarily nested values inside a structure.
- Backtracking happens both internally when constraints are enforced on non-deterministic matches—e.g., in our example when looking for two equal constant declarations—and externally, where a user can use **fail** in a target expression to rollback the state and trigger another possible match in a non-deterministic pattern or fall through.
- There are various looping constructs including the generalized **for**-loop used in our example that supports pattern matching iteration, a **solve**-loop that finds a fixedpoint of an operation over a set of variables, and fixedpoint-iterating traversal strategies (**innermost** and **outermost**).

This power of expressiveness unfortunately comes with a heavy price: the semantics of these languages becomes increasingly hard to reason about. There exist few automated tools that can handle validation and verification of transformation programs, and those that exist are limited in expressiveness of language constructs or supported properties. They can largely be divided in three categories:

1. *Black-box* techniques—e.g., Claessen and Hughes [2000], Yang et al. [2011], Finot et al. [2013] and Mongiovi et al. [2014]—rely solely on input and output specifications to validate the transformation programs. These techniques are independent of transformation program definitions which eases reuse for new languages, but can rarely provide a complete verification and can potentially miss a large amount of internal logic that is left unvalidated.

2. *White-box* techniques—e.g., Jackson et al. [2011], Büttner et al. [2012], and Clarisó et al. [2016]—that use the definitions of program definitions, but are limited in the the expressiveness of language constructs they handle (non-Turing complete) and thus do not support more interesting transformation programs such as refactorings or optimizations. Furthermore, many rely on ad-hoc encoding of the complete transformation programs to a logical formula to use in a solver (e.g. SMT or CLP) which is a process that is error prone, and hard to extend because there is a lack of explicit representation of the program state. Those that do use static analysis-based—e.g., Cuadrado et al. [2017]—techniques seem to scale better with regards to language expressiveness and properties, but currently only focus on simple type and structural properties and not on more complex properties such as abstraction of data (integers, strings, etc.), inductive properties or shapes of collections.

3. *Translation validation*-based techniques [Samet, 1976; Pnueli et al., 1998; Rival, 2004] can automatically prove sophisticated semantic properties about object programs, but only work for individual runs of a transformation. The strong guarantees they provide however are only for the provided input programs at each run, and no guarantees are provided on other inputs. This entails that errors are only detected late in the process, and fixing the errors at that stage can be very costly and time consuming.

The existing work therefore leaves space for improvement with regards to validating and verifying fully-expressive transformation programs effectively using their definitions.

1.2 Problem Definition

The void left by the existing work has consequences that go beyond mere technical curiosity. The availability of quality assurance techniques (including validation and verification) is a key characteristic for the trustworthiness of a programming language and paves way for adoption of the language in use of critical software systems. This is especially true of modern critical systems which are steadily increasing in complexity and it is here the aforementioned transformation languages would really shine in increasing readability and maintainability of such systems.

What particularly motivates this thesis is that despite the rich applied tradition of automated formal validation and verification methods in programming language research—deductive methods, abstract interpretation, symbolic execution, model checking—we paradoxically lack tools for quality assurance of transformation languages. If we are able to use these foundational techniques, we would benefit from the existing wealth of experience, be able to more effectively verify target transformations, and allow extending the techniques towards supporting properties more relevant to these languages—pertaining to typed syntactic structures and input/outpt relations—in ways more actively exploiting the information provided by the high-level features.

In summary, the problem statement addressed by this dissertation is:

> *How is it possible to extend automated formal methods from traditional programming language research to scale to handling the expressive features found in realistic high-level transformation languages?*

Concretely, the problem statement will be addressed by addressing each of the following subproblems:

P1 Unclarity regarding expressiveness of available features in model and graph transformation languages. While some high-level transformation languages like TXL and Rascal are clearly programming languages with features targeting transformations, other languages that focus on transforming models or graphs like ATL and

ETL have a more declarative feel with limited use of explicit loops and recursion. As a result, it is unclear whether these languages in general are able to express general computation (Turing-complete), and whether it is possible to exploit the lack of expressiveness to provide better analysis methods. Clarifying this problem also clarifies whether the existing techniques that handle limited subsets are sufficient for handling more interesting transformations that have more complete use of available language features.

P2 Available techniques for systematic testing of transformations do not effectively exploit definition of transformation programs Traditionally, techniques that exploit program definitions (*white-box*) often generate better test cases than those that only use input specification (*black-box*), and I believe that this is also the case for transformations. This problem therefore primarily concerns how to extend existing effective white-box techniques to handle the language constructs and derived constraints present in fully-expressive transformation languages. This is further complicated by the fact that transformations manipulate rich typed data structures—containing a wealth of different types, that each are often both deep (many nested containments) and wide (many subcomponents)—and collections of these structures and so any naive algorithm will easily hit a combinatorial explosion.

P3 Lack of formal techniques for automated verification of type and shape properties for fully-expressive transformations Most interesting transformations like normalization procedures, refactorings and optimizations require the full expressive power of the high-level transformation languages to be encoded. Existing techniques which encode the complete transformation as a logical formula for an SMT solver (or similar systems) are unlikely to scale to this approach since their employed techniques are limited with regards to loops and complex recursion and bounded model finding approaches do not scale for large systems. Conversely, techniques like translation validation are useful for verifying complex properties but only work for individual runs of a transformation. This problem thus regards finding a technique for verifying transformations that i) runs automatically ii) is easily extensible, and iii) works for all executions of a transformation.

Importantly, the technique should be able to verify interesting properties about the transformations themselves, i.e., properties that relate to the typed abstract structures manipulated by the transformations such as types and shapes relating input and output values.

1.3 Aim and Scope

The primary aim of this thesis is to develop foundational theories and tools for high-level transformation languages based on traditional programming language techniques.

This aim will be fulfilled by satisfying the following objectives:

O1 Identify key features of high-level transformation languages, classifying the languages according to their computational expressiveness The purpose of this objective is two-fold i) it aims to set the context for objectives O3 and O4 by identifying the key features that should be handled by the respective analyses, and ii) it addresses problem P1 by clarifying the expressiveness of declarative transformation languages. I partly reach this objective by systematically investigating the documentation and relevant primary and survey papers about a representative selection of high-level transformation languages, particularly focusing on declarative model and graph transformation languages to clarify their expressiveness. Formal proofs of expressiveness are deemed out of scope, since most languages are not formally specified and it is both infeasible to exhaustively try to combine available features in a variety of distinct languages to check their expressiveness and may also be practically useless if relying on combinations of uncommon constructs. To get a first-hand grasp on these features, I use a selection of the languages to encode realistic transformations and specify the semantics for core high-level transformation language constructs in a way that makes it possible to formally reason about them.

O2 Validate the semantic preservation for an industrial-scale transformation using translation validation Having identified key features of high-level transformation languages in O1, this objective focuses on providing first-hand experience with understanding and validating industrial-scale transformation programs.

This serves two purposes: it pinpoints the language constructs used in realistic transformations, providing a clearer direction for objectives O3 and O4, and it provides a deeper understanding of the strength and limitations of translation validation-based approaches.

O3 Develop a symbolic executor that supports identified high-level language features for effective automatic testing of transformations This objective focuses on addressing problem P2 by developing a technique that is able to effectively generate test cases using the source code of input transformation programs.

The approach I take is to model identified key transformation features in a small formal language, and then use symbolic execution to run the transformation generating constraints representing program paths which can be used to generate test cases triggering each explored path. The main challenge here is to accurately track constraints on typed structured input during execution of the available expressive language constructs such as deep traversal in a sound way, while still can be provided to a solver to eliminate infeasible path during execution and generate relevant test cases.

O4 Develop a formal abstract interpreter that supports verifying type and shape properties for high-level transformation languages Using the experiences gathered with the symbolic executor in objective O3, I will go beyond testing of high-level transformations and focus on verification of type and shape properties. I use Schmidt-style abstract interpretation [Schmidt, 1998] to build a sophisticated static analysis that supports verifying type and shape properties for a large subset of a realistic high-level transforation language like Rascal, which both requires handling complex control flow present in these languages—including various control operators, backtracking and exceptions—and intelligently adapting the core idea of memoization—a form of widening on the traces of execution that the traces are finitely representable—to the various looping constructs and recursion while still preserving necessary precision.

1.4 Contributions

Core Contributions

This thesis realizes the stated objectives by the following series of contributions:

C1 Systematic study fulfilling objective O1 showing that all surveyed declarative model and graph transformation languages, except bidirectional ones, are as expressive as ordinary programming languages (Turing-complete). This is significant because it shows that existing techniques that target less expressive subsets do not cover the complete target languages, and are unlikely to be able to handle more expressive transformations.

Technical Report Ahmad Salim Al-Sibahi. On the computational expressiveness of model transformation languages. *IT University. Technical Report Series*, January 2015. ISSN 1600-6100

C2 An application of translation validation to a large industrial software modernization transformation written in the high-level language TXL converting an imperative configuration system in C++ to declarative constraints. This provided a deeper understanding of practical use of available high-level transformation constructs fulfilling objective O2.

Conference Paper Alexandru Florin Iosif-Lazăr, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wąsowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA*, pages 597–607, 2015. doi: 10.1109/ASE.2015.84

C3 Symbolic execution technique with fully formal semantics supporting high-level transformation language features such as deep type-directed querying, sets, and containment instantiated for a small IMP-like transformation language called TRON. This technique fulfills objective O3 showing that it is possible to extend symbolic execution to support high-level features by state of the art techniques like lazy initialization for pointers and symbolic set constraints with new innovation like deep containment constraints that provide access to all elements of a particular type reachable for a given value abstracting over the intermediate structure. The technique is compared experimentally of a series of realistic transformations against a baseline black-box test generator optimizing towards literature-recommended metrics, showing that it is possible to achieve a significant improvement in coverage for most evaluation subjects.

Conference Paper Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wąsowski. Symbolic execution of high-level transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands*, pages 207–220, 2016. doi: 10.1145/2997364.2997382

Extended Technical Report Ahmad Salim Al-Sibahi, Aleksandar Dimovski, and Andrzej Wąsowski. SymexTRON: Symbolic execution of high-level transformation languages: Symbolic execution of high-level transformations. *IT University. Technical Report Series*, September 2016. ISSN 1600-6100

C4 A formal semantics for a large subset of the Rascal language, called Rascal Light, suitable for developing formal analyses of realistic high-level transformation programs. This represents the first formalization of a large subset of Rascal, which can be useful both for the Rascal community as a way to increase trust in the implementation, and high-level transformation language verification researchers who can use it as a basis for building their verification tools and techniques. The subset supports all relevant high-level features including the traversal construct **visit** with all available strategies, the generalized pattern matching **for**-loop, general looping constructs **while** and **solve**, the expressive pattern matching language with collection matches, deep matching and non-linear matching, control operators including backtracking and rollback, and exceptions. The semantics is based on the available documentation, testing of constructs, and expert contributions from the Rascal developers; I show the soundness of the semantics by proving a series of core theorems.

Technical Report Ahmad Salim Al-Sibahi. The Formal Semantics of Rascal Light. *CoRR*, abs/1703.02312, 2017a. URL `http://arxiv.org/abs/1703.02312`

C5 A Schmidt-style abstract interpreter derived from the formal operational semantics of a large relevant subset of the high-level transformation language Rascal supporting verification of inductive type and shape properties. This contribution fulfills objective O3 contributing theoretical insights including i) a modular domain construction that can be used to abstract individual types of values in languages operation on large domains like Rascal, ii) adaptations of trace memoization (core technique of Schmidt-style abstract interpretation) to various looping constructs, traversals and recursion using available information by each construct to maintain precision, and iii) a representation of state that has a first–class representation of control-flow operations to ensure a close correspondence between abstract and concrete semantic rules. The technique is evaluated on a series of real and realistic transformations and is able to show properties that include that a desugaring correctly translates sugared constructs, that a normalization procedure produces an output in normal form and a rename field refactoring produces programs that do not reference the old field name.

Manuscript Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Thomas P. Jensen, and Andrzej Wąsowski. Verifying Transformations using Inductive Shape Analysis. 2017[1]

Other Scientifc Activity

I have contributed to a series of papers on efficiently model checking program families by abstracting variability, which I reference below. The papers are indirectly related to my core objectives, since they use transformations to convert a process model with variability to a small set of process models where the variability has been abstracted away, making them useable with off-the-shelf model checkers.

- **Conference Paper** Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-based model checking without a family-based model checker. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, pages 282–299, 2015a. doi: 10.1007/978-3-319-23404-5_18

[1]Author ordering to be determined, currently alphabetical.

- **Journal Paper** Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Efficient family-based model checking via variability abstractions. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2016. ISSN 1433-2787. doi: 10.1007/s10009-016-0425-2. URL `http://dx.doi.org/10.1007/s10009-016-0425-2`

Related to the papers above, I have held an invited talk at the 2015 Workshop on Software Product Line Analysis Tools (co-located with SPLC 2015) on our joint work:

- **Presentation** Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-based model checking using off-the-shelf model checkers: extended abstract. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, page 397, 2015b. doi: 10.1145/2791060.2791119. URL `http://doi.acm.org/10.1145/2791060.2791119`

In relation to my work on abstract interpretation of high-level transformations, I presented a poster at the Student Research Competition at POPL 2017. The poster presented work on shape abstractions for TRON, which was used as a stepping stone to C5.

- **Poster** Ahmad Salim Al-Sibahi. Abstract interpretation of high-level transformations. In *Student Research Competition at 44th ACM SIGPLAN Symposium on Principles of Programming Languages 2017, POPL SRC 2017, Paris, France, January 15-21, 2017*, 2017b. Poster

1.5 Overview

This thesis has seven further chapters explaining my stated contributions in detail. Chapter 2 provides background on high-level transformation languages and Chapter 3 summarizes our survey on expressiveness of a wide range of declarative model and graph transformation languages (C1). Chapter 4 summarizes the experiences behind validating a realistic modernization transformation (C2), and Chapter 5 summarizes a formal symbolic execution algorithm with support for high-level transformation language features (C3). The formal semantics of Rascal Light (C4) is presented in Chapter 6, and shape-based abstract interpreter (C5) is presented in Chapter 7. Finally, Chapter 8 concludes my results.

Chapter 2

High-Level Transformation Languages

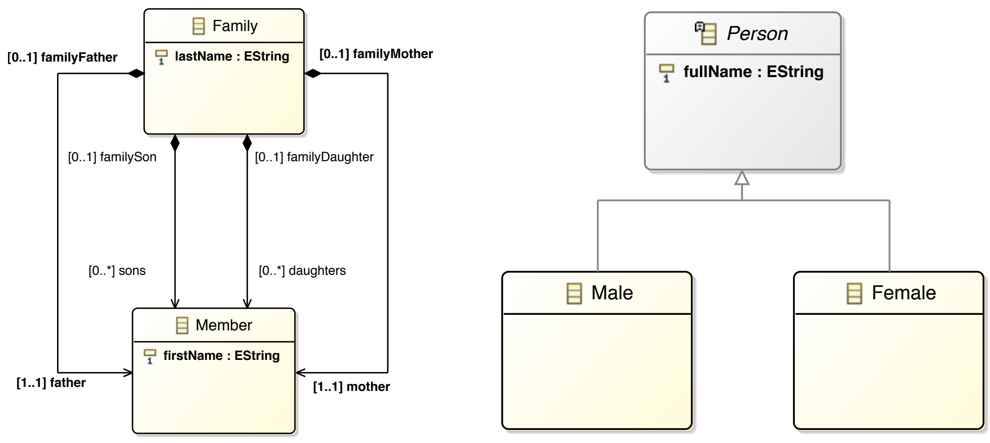
Taking into consideration the ubiquity of transformations in all aspects of software engineering, it is no surprise that there exist various transformation languages across different communities each specialized to sets of relevant tasks. In this chapter, I summarize these approaches and discuss the available features and what makes them unique.

2.1 Declarative Transformation Languages

Model transformation [Mens and Gorp, 2006; Czarnecki and Helsen, 2006] is one of the most central techniques employed in the field of model-driven engineering. A *model transformation language* allows the programmer to translate elements from one or more source models to elements in one or more target models, given their formal descriptions or *meta-models*.

Declarative model transformation languages are largely rule-based—the programmer specifies how a subgroup of elements in the source model maps to the desired target elements—and can be categorised further based on whether they mainly use graph rewriting as a formalism or not. To give a more concrete grasp on how these languages work, I will use the traditional Families to Persons [Allilaire and Jouault, 2007] transformation as a running example.

The meta-models for both source and target models for the model transformation are shown in Figure 2.1. The source Families meta-model (Figure 2.1a) models a traditional family with a common last name, which contains a mother, father, sons and daughters, each having their own first name; note, that the links are modelled bidirec-



(a) Families meta-model (b) Persons meta-model

Figure 2.1: Meta-models for the Families to Persons [Allilaire and Jouault, 2007] transformation

tionally and so each member has a reference to the containing family (`familyMother`, `familyFather`, `familySon` or `familyDaughter`). The target Persons meta-model (Figure 2.1b) models individual persons, either male or female, including their full name.

Rule-based Model Transformation Languages

Purely rule-based languages focus on providing a set of declarative rules to describe a transformation between source and target models, given their meta-models. The rules in these languages usually support rich predicate languages like OCL [Warmer and Kleppe, 1998] for constraining the scope of rule application, may allow explicit dependency between rules, and may allow the transformation to be handled imperatively for more precision. Examples of these types of languages include ATL [Jouault et al., 2008], ETL [Kolovos et al., 2008], Tefkat [Lawley and Steel, 2006] and QVT [Object Management Group, 2016].

The example Families to Persons transformation is shown implemented in ETL[1] in Figure 2.2. The transformation consists of two *helper* operations and two transformation rules. The `familyName` operation retrieves the last name for a member using the reverse link to the family that is defined—recall that a member is contained in a family and so

[1]Ported and simplified from the original ATL in Allilaire and Jouault [2007]

```
@cached
operation Families!Member familyName(): String {
  if (self.familyFather.isDefined())
    return self.familyFather.lastName;
  else if (self.familyMother.isDefined())
    return self.familyMother.lastName;
  else if (self.familySon.isDefined())
    return self.familySon.lastName;
  else return self.familyDaughter.lastName;
}

@cached
operation Families!Member isFemale(): Boolean {
  return self.familyMother.isDefined()
  or self.familyDaughter.isDefined();
}

rule Member2Female
 transform member : Families!Member
  to person : Persons!Female {
    guard: member.isFemale()
    person.fullName =
      member.firstName + " " + member.familyName();
  }

rule Member2Male
  transform member : Families!Member
  to person : Persons!Male {
    guard: not member.isFemale()
     person.fullName =
       member.firstName + " " + member.familyName();
   }
```

Figure 2.2: The Families to Persons [Allilaire and Jouault, 2007] transformation in ETL

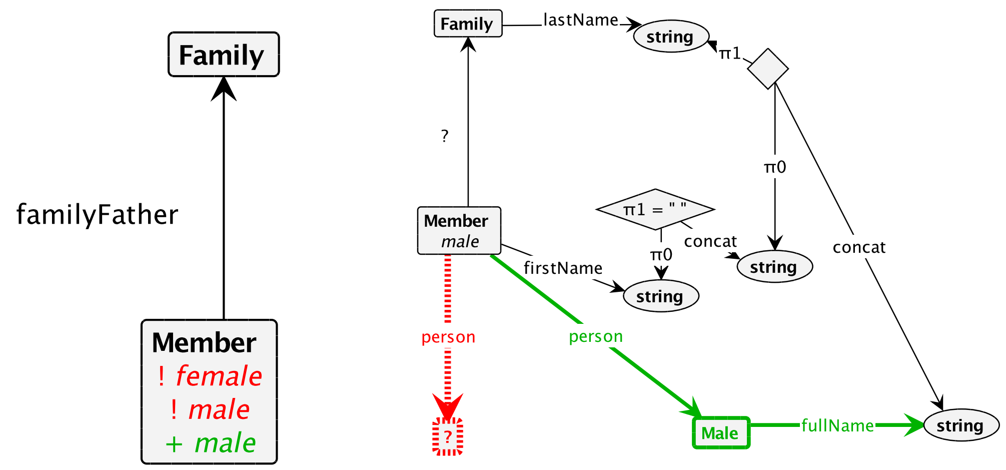
there must exist exactly one reverse link—and the `isFemale` operation determines whether a member is female (i.e., mother or daughter) or not (i.e., father or son). The `Member2Female` rule converts `Members` to `Female` persons, under the condition specified by **guard** that the input `member` is female, assigning the output `person`'s full name using the first and family names from the input `member`; the `Member2Male` works analogously but converts to `Male` persons assuming the input `member` is not female. In ETL (also ATL, QVT and similar languages) such rules are scheduled automatically, so `Member2Female` and `Member2Male` are applied on each matching element of type `Member` from the input model of type `Family` potentially producing a collection of `Persons`, either `Male` or `Female`.

Bidirectional Languages Bidirectional-languages [Hidaka et al., 2016] work similarly to other declarative transformation languages except that transformations must allow conversion back from the target model to the source model. There are various approaches to this, for example requiring reversible rules as in BOTL [Braun and Marschall, 2003] and Triple Graph Grammars [Lauder et al., 2012], the requiring well-behaved operations to extract and recompose data as in Boomerang [Bohannon et al., 2008], or using an explicit synchronization mechanisms as for example in BeanBag [Xiong et al., 2009], SyncATL [Xiong et al., 2007] and X [Hu et al., 2008].

Graph Transformation Languages

Graph-based languages represent the source and target models using graphs—optionally having types or attributes—and describe transformations using sets of conditional graph rewriting rules. The graph rewriting rules specify how subgraphs in the source map to subgraphs in the target, and executing a transformation is done by applying the rewrite rules using some strategy for rule scheduling. Some graph-based languages like PROGReS [Schürr, 1994], GReAT [Balasubramanian et al., 2007], GrGen.NET [Jakumeit et al., 2010], Motif [Syriani and Vangheluwe, 2013], VIATRA2 [Varró and Balogh, 2007], VMTS [Levendovszky et al., 2005], SDM [Fischer et al., 2000] and GROOVE [Rensink, 2003] use a separate imperative control language to let users define rule scheduling strategies, while other languages like AGG [Taentzer, 2010] and Atom3 [de Lara et al., 2004] instead use a fixed strategy that applies rules as long as possible, but allow some form of *layering* that allows ordering of rules.

Two graph rewriting rules in GROOVE for the Families to Persons example transformation are presented in Figure 2.3. The rule in Figure 2.3a adds a label *male* (highlighted using `+` and is green colored) in case the target member has a `familyFather` link defined and is not already tagged by a *female* or *male* label (negative conditions are highlighted with `!` and are red colored). The rule in Figure 2.3b corresponds to the `Member2Male` rule in ETL, and translates a family member tagged with a *male* label to a corresponding new `Male` person—if not already translated as indicated by the negative condition—with the full name being the concatenation of the first name and last name of the target member (operations are defined as subgraphs which rewrite to produce the correct value). This style of transformations is comparatively more visual than the purely rule-based style by ETL, but at the expense of be-



(a) Labelling fathers as male

(b) Translating male family members to persons

Figure 2.3: Two graph rewriting operations in GROOVE for the Families to Persons transformation

ing more low-level with regards to changes in the graph since additions and deletions of nodes and edges are done more explicitly.

2.2 Program Transformation Languages

The line between model and program transformation languages is fluid, but in general specialized program transformation languages focus more on transformation of syntax—including dealing with concrete syntax and parsing—and try to provide more expressive language features such as supporting higher-order transformations. I will use the Negation Normal Form transformation (NNF, see Figure 2.4) as a running example to present some of the different kinds of program transformation languages. The transformation converts propositional formula (Figure 2.4a)—containing atoms, negation, conjunction, disjunction and implication—to an equivalent formula where all negations are on atoms and without implications (Figure 2.4b).

Rewriting Languages

Rule-based program transformation languages like TXL [Cordy, 2006] are similar to rule-based transformation languages in that they represent a transformation as a set of rules that are scheduled automatically

$$\phi, \psi ::= x \in \text{Atom} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi$$

(a) Formula syntax

$$l ::= x \in \text{Atom} \mid \neg x \qquad \phi_{\text{nnf}}, \psi_{\text{nnf}} ::= l \mid \phi_{\text{nnf}} \wedge \psi_{\text{nnf}} \mid \phi_{\text{nnf}} \vee \psi_{\text{nnf}}$$

(b) Normalized Formula syntax

Figure 2.4: Negation Normal Form

during execution; the key difference is that TXL transforms concrete syntax trees instead of structured models—-supporting specification of and parsing using concrete grammars as first-class constructs—and supports more expressive features like run-time re-parsing, syntactic substitution and polymorphic rules.

Languages based more directly on term rewriting [Baader and Nipkow, 1998; Cirstea and Kirchner, 2001a,b] focus more on allowing on providing rich composable rewriting using different types of evaluation strategies that include support for backtracking, recursion and term traversal. Languages that primarily follow this paradigm, include programming languages like Stratego/XT [Bravenboer et al., 2008], ELAN [Borovanský et al., 1998] and Maude [Clavel et al., 2007], and frameworks like PLT Redex [Felleisen et al., 2009] for Racket [Felleisen et al., 2015] and Kiama [Sloane, 2009] for Scala [Odersky et al., 2004]. Figure 2.5 presents the NNF transformation in Stratego/XT: formulae are represented as terms using typed constructs, and the transformation `normalize` executes the sequence of rewriting rules tagged with `NNF` using the outermost strategy, which rewrites terms in a top-down fashion until a fixed-point is reached.

General Program Transformation Languages

General transformation languages like the aforementioned Rascal [Klint et al., 2011], are ordinary programming languages that provide high-level constructs suitable for transformation such as powerful pattern matching, traversal and backtracking. Other examples include languages like Xtend [The Eclipse Foundation, 2014] and frameworks like Uniplate [Mitchell and Runciman, 2007] for Haskell [Marlow, 2010]. Figure 2.6 presents the NNF transformation in Haskell using Uniplate; the transformation uses the higher-order function `rewrite` that traverses input formula exhaustively applying function `nnf'`, which is an ordinary

```
module nnf
signature
  constructors
    Atom          : ID -> Formula
    Neg           : Formula -> Formula
    And           : Formula * Formula -> Formula
    Or            : Formula * Formula -> Formula
    Imp           : Formula * Formula -> Formula
strategies
  normalize = outermost(NNF)
rules
  NNF : Neg(Or(e1, e2))  -> And(Neg(e1), Neg(e2))
  NNF : Neg(And(e1, e2)) -> Or(Neg(e1), Neg(e2))
  NNF : Neg(Neg(e))      -> e
  NNF : Neg(Imp(e1, e2)) -> And(e1, Neg(e2))
  NNF : Imp(e1, e2)      -> Or(Neg(e1), e2)
```

Figure 2.5: Negation Normal Form Transformation in Stratego/XT

Haskell function defined by pattern matching that produces a possible result value for each applicable case.

```
data Formula = FAtom String
             | (:!:) Formula
             | Formula :&: Formula
             | Formula :|: Formula
             | Formula :=>: Formula

nnf :: Formula -> Formula
nnf = rewrite nnf'
  where nnf' ((:!:) ((:!:) phi))   = Just phi
        nnf' ((:!:) (phi :&: psi)) = Just (((:!:) phi) :|: ((:!:) psi))
        nnf' ((:!:) (phi :|: psi)) = Just (((:!:) phi) :&: ((:!:) psi))
        nnf'        (phi :=>: psi) = Just (((:!:) phi) :|: psi)
        nnf' _ = Nothing
```

Figure 2.6: Negation Normal Form Transformation in Haskell using Uniplate

2.3 Programs and Paradigms

In principle, any programming language is a transformation language, but I have in this chapter presented languages across different paradigms which have features designed primarily towards transformation of models and programs. In the chapters to come, I will focus on the programming-oriented high-level transformation languages, because these allow expressing the more interesting transformations—refactorings, normalization procedures, optimizations, translations—and align the best with my goal of adapting techniques from programming language verification. Furthermore, as we will see in Chapter 3 there is little to gain of choosing the more declarative model and graph from a verification perspective.

In particular, later in this dissertation I describe an effort validating an industrial transformation in the mentioned TXL language (Chapter 4), which provided a particularly good match for the target code base because of its powerful capabilities for manipulating C++ syntax. To get a good grasp of the verification challenges introduced by the high-level transformation features, I develop a symbolic execution algorithm (Chapter 5) for a small purposely defined formal transformation language called TRON. Finally, I scale up the verification effort focusing on a large expressive subset of the Rascal transformation languages, for which I develop a verified formal semantics (Chapter 6) and an abstract interpretation based static analysis tool (Chapter 7). Rascal is particularly suitable as a target system because it provides a representative selection of high-level transformation language features—traversal, backtracking, fixed-point iteration—modelled explicitly, has an available up-to-date open source implementation,[2] and plenty of realistic programs that can be used for evaluation.

[2] https://github.com/usethesource/Rascal

Chapter 3

Expressiveness of Declarative Model Transformation Languages*

The declarative model transformation languages presented in Chapter 2 provide a convient rule-based interface for translating between various models. It is however unclear that the computational capabilities [Taylor, 1998] of such languages matches those of general programming languages, since modification of state can be limited and repetition can be implicit or bounded. The computational expressiveness of a language matters from a verification point of view, since a large variety of techniques only apply on languages with limited computational expressiveness. This chapters presents a systematic analysis of a wide selection of declarative model transformation languages, seeking to pinpoint the constructs that enables or limits the level of expressiveness.

3.1 Surveys on Model Transformations

Language Expressiveness To the author's knowledge there has been no overall study on computational expressiveness of model transformation languages, but there do exist several studies that compare specific aspects. Gomes et al. [2014] compare a variety of languages on their pattern matching capabilities in order to clarify the performance implications of each system[1]. Different language features of ATL and SDM are

[1]Additionally, a source of inspiration for selection of transformation languages.

*This chapter is based on Al-Sibahi [2015], which is published as a technical report.

compared in a case study [Patzina and Patzina, 2012], presenting a table clarifying the capabilities of these languages, including support for recursion and in-place transformation. Finally, several model transformation language implementation papers [Varró and Balogh, 2007; Syriani and Vangheluwe, 2013] present small comparisons of language features; however, these do not provide an analysis of language expressiveness, and are neither comprehensive in their comparisons.

General Mens and Gorp [2006] provide definitions for kinds of model transformations, and classifies them according to several characteristics including whether they transform across different abstraction levels, between different meta-model definitions and are primarily syntactic or semantic. Czarnecki and Helsen [2006] provide a detailed classification of a large selection of model transformation languages according to the set of available features including constraint and transformation rule specifications, level of control over execution and modularity; Hidaka et al. [2016] provides a similar feature-oriented classification for bidirectional transformations. Several surveys [Amrani et al., 2012; Calegari and Szasz, 2013; Rahim and Whittle, 2015] focus on summarizing the state of the art in validation and verification of model transformation, classifying according to the techniques used, transformation features and properties supported. All of these surveys do not address my objective (O1, Section 1.3) which is to identify to identify the computational expressiveness, except Hidaka et al. [2016] which does so only for bidirectional languages.

3.2 Computational expessiveness

Computability is usually defined in terms of the Church-Turing thesis [Kleene, 1967; Copeland, 2015] which states that any *effectively computable* mathematical function can be computed by a Turing machine [Hopcroft et al., 2007]. A language is said to be Turing-complete if it is as expressive as a Turing machine, in the sense that it can compute the same any function a Turing machine can compute. Known examples of Turing-equivalent systems include the λ-calculus [Barengdt, 1985; Alama, 2014] and term-rewriting systems [Post, 1947].

While it can be hard to pinpoint the combination of features needed for a language to be Turing-complete, we can specify some general characteristics:

State change It should be possible to continue computation with a new state, whether by writing on the tape as the Turing machines, by reduction as in the λ-calculus or by subterm replacement as in term rewriting.

Branching It should be possible to make different decisions on input, similarly to how Turing-machines conditionally move the head depending on the current symbol or how term-rewriting systems express rules conditional on patterns.

Unbounded execution time The language must not bound the number of execution steps, e.g., by providing repetition constructs such as recursion or iteration.

Unbounded memory There must be no bounds on memory, analogously to the infinite extensibility of a tape in Turing-machines and the unbounded term size in λ-calculus.

Determining the computational expressiveness for a language has been long known to be important with regards to verification. In particular, it determines the class of interesting semantic properties we can effectively decide [Rice, 1953] and thus also affects the verification techniques applicable.

General results on graph rewriting

Graph rewriting is the base formalism for many declarative model transformation languages, as mentioned in Chapter 2. Therefore, it is important to understand the expressiveness of general graph rewriting systems, and clarify the conditions necessary to achieve expected expressiveness results.

Proposition 3.1. *Graph rewriting systems can express Turing-complete computation.*

Proof. Shown by Plump [1998]. □

Existing research shows that graph rewriting systems can express Turing-complete computation. Albeit similar, graph rewriting systems are richer than term rewriting systems—possibly having cycles and shared nodes—and do not share the same termination citeria; Plump [1998] shows it is possible to translate a non-terminating term rewriting systems to a terminating graph rewriting systems and vice versa. Generally, a base requirement for graph rewriting systems to be terminating is that all the rules contained must either delete nodes or have application conditions [Plump, 1995; Bisztray and Heckel, 2010]. Any other graph rewriting system is trivially non-terminating since the same rule can be reapplied ad infinitum.

3.3 Analysis Method

The formal process to (dis)prove that a language is Turing-complete is by reduction, so to show that it is (im)possible to encode another Turing-equivalent system or an undecideable problem—e.g., Post correspondence problem [Post, 1946]—in the language. It is however practically infeasible to do such a formal proof for each of the various languages examined, since it requires manually setting up each system, learning the language sufficiently to perform a complete analysis, and verifying that such analysis is semantically correct.

Instead, I perform a literature-based analysis whereby I use published articles, language manuals and other related documentation to check whether there is a combinations of features that make a language Turing-complete. I will use the (semi-)formal semantics of target language as a primary source if provided, but in most cases I use a combination of sources to establish an overview of the target language capabilities. The results provided should be sufficiently accurate, and reflect better the features that ordinary users meet.

Collection process

The collection process proceeded as follows:

- If the language documentation explicitly mentions or proves that the language is Turing-complete, I note down the result and language features required to achieve such expressiveness.
- Otherwise, I search the documentation for whether there is a combination of constructs that make the target language Turing-complete.

The constructs required for Turing-complete computation differ across languages examined. For the subset of languages that directly allow use of imperative constructs, it is straightforward to check the expressiveness. If the target language supports creation and processing of variably-sized data, conditionals and unbounded iteration or recursion, then it is definitely Turing-complete.

For primarily rule-based transformation languages with implicit control flow, the problem is more subtle. The analysis must consider possible limitations of rule definition and execution, and check whether non-terminating rules are rejected or iteration is only allowed on bounded collections.

3.4 Analysis of computational expressiveness

Paradigms The investigated languages are summarized according to supported paradigms in Table 3.1. All investigated declarative model transformation languages are unsurprisingly rule-based, except the operational subset of QVT and ATC which is a low-level implementation oriented language. The investigated languages are divided on the underlying formalism, with more than half the languages being primarily graph-oriented and the others being oriented towards metamodels. Furthermore, I have chosen to include two primarily bidirectional languages—BeanBag and BOTL—for comparison, although a more comprehensive analysis of expressiveness of bidirectional languages is available in Hidaka et al. [2016]. Finally, Table 3.1 shows a division regarding imperative constructs such as state updates and loops, spreading evenly between languages that do and do not support such constructs.

Expressiveness The expressiveness of investigated languages is presented in Table 3.2, which summarizes available repetition constructs and whether each language is Turing-complete. The repetition constructs supported are varied across different languages, but I divided them in three main categories according to whether the investigated languages support explicit loops like `for` and `while`, whether they support some form of recursion and whether they have implicit repetition in the execution engine, so that execution of the specified rule set continues until a fixed-point is reached.

All presented languages except the bidirectional ones are found to be Turing-complete based on the detailed analysis in Section 3.4; this is consistent with the results by Hidaka et al. [2016] which argue that most bidirectional languages are by construction non-Turing complete, since they enforce functional relationships. Two of the presented languages (AGG and VMTS) notably support checking termination of transformations, but these checks are optional and not enforced.

Detailed analysis

This section discusses the details of the results presented in Table 3.1 and Table 3.2. It provides a summary for each langauge, whether the language is Turing-complete and what combination of features contribute to the expressiveness.

Language	Rule-based			Imperative
	General	Graph-oriented	Bidirectional	
PROGReS	✓	✓	✗	✓
AGG	✓	✓	✗	✗
GReAT	✓	✓	✗	✗
GrGen.NET	✓	✓	✗	✓
Motif	✓	✓	✗	✗
Atom3	✓	✓	✗	A01
Tefkat	✓	✗	✗	✗
QVT Rel.	✓	✗	✓	QR1
QVT Op.	✗	✗	✗	✓
BOTL	✓	✗	✓	✗
BeanBag	✓	✗	✓	✗
VIATRA2	✓	✓	✗	✓
VMTS	✓	✓	✗	VM1
ATL	✓	✗	✗	✓
ETL	✓	✗	✗	✓
SDM	✓	✓	✗	✓
ATC	✗	✗	✗	✓

A01	Constraints are specified by python statements
QR1	Can call QVT Operational mappings
VM1	Transformation on attributes using XSLT

Table 3.1: Paradigms of transformation languages

PROGReS The language supports specification of arbitrary graph rewriting rules, whose execution is controlled by an imperative language [Schürr, 1994; Patzina and Patzina, 2012; Rozenberg, 1997]. The language is Turing-complete by Proposition 3.1 since the imperative control subset has an unbounded looping construct `loop` [The PROGReS Developer Team, 1999], and it can therefore simulate a general graph rewriting system.

AGG Arbitrary graph rewriting rules are supported in AGG [Taentzer, 2004; Runge, 2006], and the rule interpretation engine only terminates

Language	Repetition			Turing-complete	
	Looping	Recursion	Implicit		
PROGReS	✓	✓	✗	✓	
AGG	✗	✗	✓	✓	!
GReAT	✗	✓	✗	✓	
GrGen.NET	✓	✓	✗	✓	
Motif	✓	✓	✗	✓	
Atom3	✗	✗	✓	✓	
Tefkat	✗	✓	✓	✓	
QVT Rel.	✗	✓	✓	✓	
QVT Op.	◗	✓	✗	✓	
BOTL	✗	✗	◗	✗	
BeanBag	◗	✓	✗	?	
VIATRA2	✓	✓	✗	✓	
VMTS	✗	✓	✗	✓	!
ATL	◗	AT1	◗	✓	
ETL	ET1	ET1	◗	✓	
SDM	SD1	SD1	✗	✓	
ATC	✓	✓	✗	✓	

◗	Bounded number of steps
?	Status generally unknown
!	Optional termination check
AT1	Using global helpers or lazy rules
ET1	Using Epsilon Object Language (EOL)
SD1	By setting up the required control flow and path expressions

Table 3.2: Expressiveness of transformation languages

when no rule is further applicable. The language can check whether a given rule-set of a program is within the terminating fragment, but since this check is not enforced the language is considered Turing-complete by Proposition 3.1.

GReAT The language [Balasubramanian et al., 2007; Agrawal et al., 2003] supports defining transformation rules on graphs modularly as blocks that can wired together, even recursively where the output of a block can be wired as input to a block further back in the transformation sequence. Since a rule sequence is only terminating when no further output is produced, the language is Turing-complete.

GrGen.NET The language user manual [Blumer et al., 2010] directly shows an implementation of a Turing-machine. Notably, the languages works on graphs, supports recursive rules [Jakumeit, 2008], unbounded looping and general imperative constructs.

Motif The language [Syriani and Vangheluwe, 2013] transforms graphs and supports unbounded looping (FRule, SRule), backtracking and recursion (XRule), and it explicitly mentioned that it is possible to construct non-terminating programs. Therefore, the language can simulate a general graph rewriting system and is thus Turing-complete by Proposition 3.1.

Atom3 The framework uses graph-based meta-modelling [Vangheluwe et al., 2002], and transformations are specified using graph rewriting rules. The execution engine (Graph Rewriting Processor) [Levytskyy and Kerckhoffs, 2003; de Lara et al., 2004] is not limited and applies rules to exhaustion, making the language Turing-complete per Proposition 3.1. Remarkably, Motif is implemented in Atom3 which further strengthens the argument for Turing-completeness.

Tefkat This transformation language [Lawley and Steel, 2006] is based primarily on the second revised submission report on QVT [Duddy et al., 2004], which explicitly touts Turing-completeness. Transformation rules can specify expressive constraints on *both* source and target models, and it supports unbounded recursion.

QVT Relations The language [Object Management Group, 2016] is Turing-complete since it can simulate general graph rewriting systems by in-place transformations, where transformation rules (relations) are re-applied until all required constraints are satisfied. Additionally, it is possible further call imperative rules from the operational subset and arbitrary external code using the QVT Blackbox mechanism.

QVT Operational The operational subset of QVT [Object Management Group, 2016] is Turing-complete since it is possible to specify recursive rules [Kraas, 2014], in combination with creation of intermediate data, branching and looping. This makes it similar to many general-purpose imperative programming languages.

BOTL Model transformations in BOTL [Braun and Marschall, 2003] are bounded by the number of rules defined and only applied to finite sets of matches. Therefore, all transformations are known to be terminating and the language is not Turing-complete.

BeanBag The language [Xiong et al., 2009] supports synchronisation of two models by using expressions containing various equational constraints, including variable binding, bounded iteration (`forall`, `exists`) and recursion. All the repetition constructs must however satisfy a stability property—ensuring there always exists a valid synchronisation for given models—which limits the expressiveness of these constructs; the only way to construct non-terminating programs in the language is by using counter-intuitive circular equational constraints, and most program in practice always terminate. It is unclear whether it is possible to use circular constraints intuitively to perform arbitrary computation without breaking the stability property, and without further formal analysis it can not be decided whether the language is Turing-complete or not[2]. This formal analysis is left as future work, but the paper [Xiong et al., 2009] presents a formal semantics which could be used as a starting point.

VIATRA2 The language [Varró and Balogh, 2007; Balogh and Varró, 2006] is based on two formalisms: graph transformation (GT) which

[2]Confirmed by personal correspondence with the main author, Dr. Yingfei Xiong.

allows the user to specify graph rewriting rules and abstract state machine (ASM) [Gurevich, 1995; Börger and Stärk, 2003]—a known Turing-complete formalism—that can be used to control the flow of execution. Notably, it is also possible to simulate a general graph rewriting system by using GT and the unbounded `iterate` construct from ASM, and so Proposition 3.1 applies.

VMTS This language [Levendovszky et al., 2005] is Turing-complete by Proposition 3.1 since it supports graph rewriting using a combination of visual model processors (VMP) and the visual control flow language (VCFL) [Lengyel et al., 2007]. Attribute transformations are performed using XSLT [Kay, 2017], which is known to be Turing-complete as well [Onder and Bayram, 2006].

ATL Ordinary ATL rules [Jouault et al., 2008; Troya and Vallecillo, 2011] cannot be recursive and are applied only a finite amount of times by the execution engine [Wagelaar et al., 2014]. Lazy rules [Tisi et al., 2011] can however be recursively called, which in combination with the complex constraints on source and target models makes the declarative part of ATL Turing-complete, similarly to Tefkat.

It is possible to call imperative OCL helpers from ATL rules, but all imperative constructs are bounded on the size of collections [Cengarle and Knapp, 2004]. OCL does however support recursion in helpers, and so Turing-complete computation could still be done that way.

ETL Epsilon Transformation Language [Kolovos et al., 2008, 2014] supports both ordinary and lazy rules, making the declarative part Turing-complete like ATL. Furthermore, it relies allows use of imperative constructs from the Epsilon Object Language (EOL), which includes unbounded loops (both `for` and `while`), branching and assignment.

SDM Story Driven Modelling (SDM) [Fischer et al., 2000] uses graph transformation as an underlying mechanism, where *story patterns* are used for describing modifications to models, similarly to graph rewriting rules. The patterns are executed via a control flow language that allows unbounded recursion [von Detten et al., 2012; Patzina and Patzina, 2012], and is therefore capable of expressing Turing-complete computation.

ATC As a low-level language [Estévez et al., 2006] for implementing model transformation engines, ATC supports many necessary primitives for model querying, matching, manipulation and transformation. In addition it contains common imperative language constructs, including unbounded `while` loops and is therefore Turing-complete.

3.5 Expressiveness and Verification

Eventhough the investigated languages focused more on providing a high-level declarative constructs for transforming models, the analysis presented in this chapter showed that virtually all the presented languages are very expressive and able to perform Turing-complete computation; the only notable exceptions were the bidirectional languages, BOTL and BeanBag, which were more limited by design.

This seems to indicate that transformation is a task that inherently requires a high level of expressiveness. Even a languages like ATL which typically bounds rule application and iteration, allows recursion via lazy rules and OCL helpers, and languages with termination checkers like AGG and VMTS chose to not strictly enforce termination.

These findings suggest that we can not treat declarative model transformation languages specially with regards to verification when we are interested in supporting the full feature set available. In particular, existing techniques that only support limited expressive subsets of transformation languages will only work in very specific cases, and which avoid features that are useful for more interesting transformations like refactoring.

General purpose high-level transformation languages like TXL and Rascal are known to also be Turing-complete, but provide several advantages from a verification point of view. Their data structures and basic control operations are similar to ordinary programming languages, so it is possible to more directly apply verification techniques such as symbolic execution and abstract interpretation from these communities. Furthermore, while they similarly to many of the declarative transformation languages, support high-level transformation features—backtracking, traversal and fixed-point iteration—they in contrast use an explicit execution model which is easier to reason about statically.

Chapter 4

Validating a Software Modernization Transformation[*]

Joint work with: Alexandru F. Iosif-Lazăr, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, Andrzej Wąsowski

Software modernization [Seacord et al., 2003] aims to tackle the challenge of growing size and complexity of legacy code by restructuring it to use improved technologies while preserving core functionality. To date, there is relatively little literature available on software modernization projects in practice, especially in the safety critical domain. We present experiences from an industrial software modernization project that is done in collaboration between IT University of Copenhagen and Danfoss Power Electronics,[1] a global producer of components and solutions for controlling electric motors. The modernization project is automated using a non-trivial code-to-model transformation written using the high-level rule-based transformation language TXL [Cordy, 2006], and has been particularly important for shedding light on the problems that occur in transformations, and the necessity of developing formal verification methods and tools.

[1]`http://www.danfoss.com`

[*]This chapter is based on Iosif-Lazăr et al. [2015] which is published a peer-reviewed conference publication. The primary experiments, which concern validating the transformation, were performed by the main author Alexandru F. Iosif-Lazăr based on an implementation that is done in collaboration between a separate team at ITU (Rolf-Helge Pfeiffer, Aleksandar Dimovski and Andrzej Wąsowski) and Danfoss. I participated in studying and classifying bugs in the transformation and contributed to the observations and formal justification in the paper. The gathered experience from this paper had an inspirational role for the rest of the dissertation.

Concretely, the modernization project involves a *configuration* tool used to adapt Danfoss frequency converters to a particular application. The configuration tool consists of thousands of C++ functions for accessing and validating the configuration parameters. The code base is being modernized so each configuration parameter function must be converted from the imperative C++ implementation to a declarative logical specification, so that the configuration software can be managed by an off-the-shelf constraint solver. Each existing function computes a result—either the value of a particular parameter or a Boolean value representing a check of accessibility and validity of parameters—and the expected declarative specification should be an expression or logical constraint that preserves the same result. The code modernization is done the automated transformation on each function individually; the transformation performs a sequence of syntactic replacements, gradually eliminating C++ pre-processor directives, local variables, C++ control statements and leaving behind a pure (side-effect free) expression.

TXL only ensures the syntactic correctness of transformations—enforcing grammar constraints through pattern matching—and does not guarantee that preservation of source program semantics. In the Danfoss case, determining the semantic equivalence is feasible due to the particular shape of programs: the employed subset of C++ is small with no recursion, small fixed bounds for loops, bounded input and no use of esoteric features like inline assembly.

We assessed that symbolic execution [King, 1976] is mature enough to handle this task efficiently, and implemented a *lightweight* wrapper that compiles both input and output of the transformation and employs the symbolic executor KLEE [Cadar et al., 2008] to assert their equivalence for all possible input. As the result of symbolically executing the programs, KLEE produces a set of *path conditions* that represent distinct execution paths of the two programs along with their return values as symbolic formulae; if the equivalence assertion is satisfied by all paths then the two programs are equivalent, and otherwise if there is a pair of compatible input-output path conditions where the assertion fails, then it outputs the path condition pair and a counter-example containing a set of concrete assignments to inputs that trigger the assertion failure.

Our contributions are:

- Synthesis of experiences from the design of a non-trivial modernization transformation for an industrial project.

- Designing and implementing a transformation validation technique for this case study that produces concrete counter-examples when semantic equivalence is not preserved between input and output programs.
- Lessons learnt from using the validation technique in the case study, including an analysis of the kind of errors that have been identified.

To the best of our knowledge this is the first analysis of transformation errors extracted from an industrial project of this scale and complexity (4119 functions or 14502 lines of input code of which: 105 error cases were found at transformation time, 104 error cases were found at verification time and 3910 were successfully verified).

4.1 Motivating Example

We illustrate the technique for modernizing a function and validating its correctness by using an example containing a simplified and anonymized version of an actual function from the Danfoss code base.

```
Configuration config = selectedConfParameter;
Option opt = selectedOptParameter;
bool result = false;
switch (config) {
  case config1:
    if (opt == option1) result = true;
    break;
  default:
    result = true;
    break;
}
return result;
```

(a) Example imperative parameter function

```
(selectedConfParameter == config1
  && selectedOptParameter == option1)
    ? true
    : false
```

(b) Actual output constraint from modernization transformation

```
(selectedConfParameter == config1)
 ? (selectedOptParameter == option1)
 : true
```

(c) The expected output constraint

Figure 4.1: Anonymized example validation function from the Danfoss code base

The input program (Fig. 4.1a) is a function that validates the consistency of the values provided to two selected parameters (`selectedConfParameter` and `selectedOptParameter`) with respect to each other using constants `config1` and `option1`. The goal of our modernization transformation is to produce a declarative constraint that for the same inputs produces the same truth values.

Syntactically, the input program consists of a sequence of variable initializations, a `switch` statement with two cases—one containing an `if`-statement—and a `return` statement that returns the value stored in the variable `result`. The corresponding output program should be a pure C++ expression which produces the same result as the input given the same values. Running the TXL modernization transformation produces a declarative C++ expression (Fig. 4.1b) using the following steps:

1. Replacing the `switch`-statement with a nested `if-else` conditional.

2. Simplifying the conditional statement, so local variables are replaced by their assigned values and we end up with a straightforward control flow.

3. Reducing the conditional statement a ternary expression of the form `e1 ? e2 : e3`.

It is not immediately clear that the produced output (Fig. 4.1b) has equivalent semantics to the input (Fig. 4.1a), despite this being a relatively simple example. In order to check that this is indeed the case, we perform symbolic execution on the input and output programs using KLEE to gather path conditions for all possible execution paths of each program; when there exists discrepency between the path conditions of the input and output programs, KLEE reports the discrepant path as a counter-example that acts as a witness of an execution that is possible in one program but not in the other.

For example, the path condition

$$\begin{aligned} &\mathit{selectedConfigParmeter} \neq \mathit{config}_1 \\ \wedge\, &\mathit{selectedOptParameter} = \mathit{option}_1 \\ \wedge\, &\mathit{result} = \mathsf{true} \end{aligned}$$

is generated for the input program in Fig. 4.1a but not for the transformed program in Fig. 4.1b proving that these programs are not semantically equivalent. By further investigation of the counter-example, we are able to determine that the expected output program should instead be the one shown in Fig. 4.1c.

Furthermore, we were able to diagnose the trace of transformation rules applied to the input program successfully finding the rule that produced erroneous output. The erroneous rule was a rule which tried to simplify nested `if`-statements forgetting to take the `else`-branch into consideration. The validation technique thus proved itself incredibly useful in accurately tracking where the bugs occured, allowing an experienced transformation specialist to fix this issue relatively swiftly.

4.2 The Modernization Case Study

The goal of the case study is to establish the feasibility of transforming a large imperative C++ code base for a configuration tool to a declarative model, in a manner that is *automatic*, *trustworthy* and *cost effective*. Cost effectiveness is understood here as being cheaper than reimplementing the code from scratch.

The modernization project combines transformation engineering and verification research in an exploratory case study, which acts as a pilot for larger modernization activities in the same organization. The researchers have access to the legacy C++ code base and to three Danfoss engineers/architects knowledgeable about the code, the context and the use case.

The study has two key propositions: i) To establish that freely available transformation and validation tools are sufficiently mature to execute this modernization process, and ii) to explain what kind of errors might appear in transformation projects involving complex code, even if implemented by experienced model transformation developers and language specialists.

Case Description

The configuration tool code base consists of 4119 C++ functions in total. These functions encode dependencies, visibility constraints, and default values of approximately one thousand configuration parameters of a frequency converter. There is limited use of C++ features: no object-oriented aspects are used, except for member access and limited encapsulation. Functions have straightforward control flow mostly consisting of conditionals and **switch**-statements, with occasional use of bounded iteration via **for**-loops; there is no use of `goto` statements, unbounded loops or recursion. Other constructs include variable declarations and usage of local variables in arithmetic and comparison expressions, calls to pure functions, singleton members and static members (e.g., to convert physical units), and casts between different types (both C-style casts and `static_casts`).

There are 14502 source lines of code (SLOC) that need to be modernized in the pilot project, and more similarly-looking configuration tools for other products waiting for modernization afterwards. As many as 3348 of the 4119 functions are already in expression form; these do not need to be modernized, but should left intact by the modernization transformation. The remaining 771 functions have 14.47 SLOC of code on average.

The modernization to declarative logical expressions decreases the burden of maintenance by allowing configuration to be done using an off-the-shelf verifier [Felfernig et al., 2014]. The nature of the configuration project puts some strict constraints on the implementation and quality of the modernization transformation:

Automation It is infeasible to halt the normal development of the project for an extensive period of time . An automated transformation can be developed in parallel to the actively developed code base and can be efficiently run again on the updated code base in minutes, ensuring minimal impact.

Trustworthiness The configuration code contains crucial domain information and missing configuration constraints could lead to creation of unsafe configurations by customers.

Verification Background

In the general case, it is undecidable to show semantic preservation for transformations, but in practice many analysis problems appearing in engineering of real systems can be handled using incomplete and (partly) unsound [Livshits et al., 2015] methods. This is true for our particular case where most inputs have a fixed shape, and it is perfectly acceptable that there are few corner cases are manually handled during modernization.

Our approach is aligned with translation validation [Samet, 1976; Pnueli et al., 1998] in the sense that it validates concrete translations instead of the transformation tool or algorithm itself. The path conditions produced by KLEE represent the semantic framework of the compared subset of C++programs, and an SMT solver is used by KLEE to prove the equivalence of the path conditions, providing a counter example if they are not equivalent.

Research Questions and Methods

Our methodology generally follows the framework of action research [Wohlin et al., 2012]. Our research questions are as follows:

RQ1 Is it feasible to design the aforementioned transformation using off-the-shelf technology in a limited time?

RQ2 What are the main obstacles and challenges in designing and implementing the transformation?

RQ3 Can high assurance methods be used at acceptable costs to validate the transformation?

RQ4 What kind of errors are found in a transformation implemented by experts?

Questions RQ1–3 are interesting for companies and researchers looking into technology transfer in modernization projects, and the last research question is more relevant for researchers in the fields of model and program transformations. We are not aware of existing studies that provide a detailed account on the number and type of errors in realistic software transformations. This work hypothesizes the kinds of problems worth addressing, and concretely inspired the verification efforts presented in the later chapters of this thesis.

We decided to address RQ1 and RQ2 by systematically investigating the most effective way to implement the transformation, and evaluating a variety of approaches and technologies that fulfill the requirements elicited from the industry partner. A similar process was executed for investigating validation technologies to solve RQ3, where we recorded experiences during the process to report them in this paper. We have collected statistics about the effectiveness and efficiency of the transformation for RQ2, and measure the ration of false positives for RQ3, explaining why they appear when following our validation methods. Regarding RQ4, we collected counterexamples from the validation process, classified them, and qualitatively analyzed them to understand what kind of errors arise in the transformation.

Study Participants Three teams were involved in the project:

1. The industrial partner had a team of two engineers and an architect, which presented the software modernization problem, requirements and artifacts.

2. The transformation implementation team consisted of a transformation expert, a language semantics expert and a project leader; the transformation expert designed and implemented the transformation in dialog with the other members of the team.

3. The validation team consisted of an applied verification researcher, a junior PhD student in programming languages,[2] and the same language semantics expert and same project leader that were involved in designing the transformation.

Threats to Validity

Construct Validity The case was selected by the industrial partner according to the problem they wanted to solve, and the research team only had access to the previously mentioned parts of the configuration tool. It is entirely possible that the researchers could have misunderstood particular aspects of the software architecture, but given their expertise in the relevant areas, we believe that the impact of this would only be limited. Furthermore, the results are still interesting from a research point of view since transformation is of substantial complexity, and even if it was slightly misaligned with the requirements it still sheds a lot of light on pragmatics of such transformations. From the engineering point of view, the validation method provides actual counterexample for the detected errors and is easy to manually confirm any unwanted discrepancy.

Internal Validity The validation procedure has been developed in the same study in which it is evaluated, and there is certainly a risk that it had overlooked some important errors. However, the transformation and validation projects has been designed independently, and key developers of the two parts have not communicated in any significant matter; several months have passed between the end of the transformation implementation project and the beginning of the validation project. In

[2]Me

particular, the transformation had been designed with a focus on effectiveness with no thought of later verification in mind. The only minor risk of misinterpretation could be in the explanation of the programming errors that caused each type of bug, since the mapping was done by the validation team.

External Validity Limited external validity is in the very nature of an individual study, which is why we take care to describe the properties of the case in detail. In safety critical software, simple imperative code with bounded for-loops is common, and we thus believe that the findings can generalize to additional modernization projects.

Reliability The analyzed transformation errors can be biased towards the weaknesses of the particular development team. It is unlikely that the reported errors were trivial, since the involved teams were experts in model transformation and verification with more than 12 years of combined experience. Nevertheless, we report these findings only existentially—without generalizing—providing a single data point for the research space where very little evidence is available so far. There is a definite need for more studies to get a good understanding of the nature of design errors in transformative and generative programming.

4.3 Designing the Transformation

We found that implementing an automatic transformation is superior to a manual refactoring, as it allows minimizing down-time on the main development of the code base. The code to be modernized only needs to be frozen for the few minutes it takes to run the transformation and it can be evolved freely while the transformation is being implemented and tested. This is somewhat different from the standard use case for transformations which is automating repetitive tasks in conversions of data, code or models. This transformation was meant to be executed only once, but automation was key to minimize disruptions in the regular development process.

Observation 4.1. Automatic transformations allowed us to decouple and parallelize the regular development and the modernization activities.

Constructing a general transformation from imperative C++ code into a declarative form is impossible in general, but our objective was much more modest: we only needed to handle programs with a fixed shape that covers the code at hand. We settled on saving resources whenever possible by sacrificing generality and even agreed to left a small number of cases with complex fragments involving loops to be modernized manually instead of designing rules that would handle them correctly from first principles. The manual migrations can be hard-coded into a transformation as special cases, so that the transformation execution remains automatic.

Observation 4.2. Modernization is a one-time transformation, so it was beneficial to focus only at the code at hand without eagerly generalizing.

To control the lack of generality, the implementation follows a *fail-fast* programming style [Gray, 1986; Shore, 2004]: it succeeds on the inputs that it was designed for, but fails as early as possible on inputs that violate specific assumptions required (e.g., use of unsupported language elements). This was achieved by making preconditions for rules as precise as possible and writing explicit assertions when possible (where a rule is in principle applicable but does not cover all the cases). Without fail-fast programming we would have very limited trust that the transformation works correctly on hundreds of input code fragments that we were not able to inspect manually.

Observation 4.3. Fail-fast programming helped retaining quality on required inputs, safely avoiding supporting anything not necessary for the modernization project.

Initially, we considered using semantic analysis techniques to solve the task—e.g., static single assignment or type analysis—but it soon became clear that this would considerably raise the complexity of the implementation, and more importantly pollute the output with automatically generated identifiers that are unfamiliar to developers. The resulting models are not merely left as is, but are expected to be read and modified by humans afterwards. Full understanding of static semantics might only be required if one implements a general refactoring, but for known code base it seems much easier to work with syntactic transformations.

Observation 4.4. We found working with syntax directly significantly more cost effective for ad-hoc transformation tasks than relying on semantic information.

Tool Selection

Since the input language (C++) is rather complex, we understood early on that the transformation tooling should be driven by the availability of a good C++ grammar, rather than our personal experiences. Thus we considered several existing open source compiler front-ends (for instance GCC[3]), and language tools (such as Eclipse CDT[4]) for the task, but found them too challenging to use. We then considered transformation tools (see also Chapter 2) and found that both Spoofax [Kats and Visser, 2010] and TXL [Cordy, 2006] have C++ grammars, and that the latter had a better maintained grammar for the current version of the tool. We thus ended up using TXL since in addition to having a good C++ grammar, it handles grammatical ambiguities quite well and provides mechanisms for relaxing the existing grammar, making it easy to adapt to our needs. TXL is a standalone command-line tool, with few dependencies, and we got a simple proof of concept transformation working after only 4 hours of experiments with TXL.

Observation 4.5. Simplicity and integration with the languages to be transformed had a stronger influence on the selection of tools than the features or paradigm of the transformation language.

Transformation Implementation

TXL supports working with only one grammar at a time, and so the input and output syntax must conform to the same grammar. To overcome this we selected a subset of C++ expression language as our target language, since it sufficiently well captures propositional constraints over our finite set of domain variables. To handle this format we needed to relax the C++ grammar slightly to allow top-level expressions, and to preserve correctness we use a simple validation rule to that checks whether the output program is an expression in the subset of interest.

An example transformation rule for our modernization project is presented in Figure 4.2; fully capitalized identifiers refer to syntax trees. The rule matches a conditional statement without an else clause (line 3), translating it into a ternary expression (line 8) if the body statement is neither a compound statement nor another conditional (Lines 4–5); to fill in the missing branch in the ternary expression, a constant `true`

[3]`https://gcc.gnu.org`
[4]`http://eclipse.org/cdt/`

expression is used (Lines 6–7). The specified rule preconditions, in combination with the definition of the grammar and other rules, ensure that the rule is only applied when the body statement is largely consisting of a simple expression itself.

```
1  rule convert_simple_sel_stmt
2    replace [selection_statement]
3      'if '( EXP [expression] ') STMT [statement]
4    where not STMT [contains_selection_stmt]
5    where not STMT [is_compound_stmt]
6    construct TRUE_STMT [true_case_statement]
7      'TRUE ';
8    by '( EXP ') '? '( STMT ') ': '( TRUE_STMT ')
9  end rule
```

Figure 4.2: Example TXL transformation rule from the modernization project: translating a conditional statement into a ternary expression.

The overall algorithm applied by the transformation is:

1. The program fragment is checked for format assumptions: all branches return a value, there are no loops and goto jumps, no calls to non-pure methods, etc. The transformation is parametrized by a list of names of pure functions, since it does not do any semantic analysis to determine purity itself.

2. All preprocessor `#ifdef` directives in the program are cleaned up, and converted to ordinary `if` statements.

3. Local variable assignments are inlined so that variable accesses are replaced by assigned values—going from the last assignment to the first to maintain correct ordering—and corresponding declarations are removed.

4. All `switch` statements are converted to a series of `if` statements.

5. Sequenced `if` statements are simplified into nested `if` statements, such that the fragments are reduced to a single root statement.

6. `if` statements are converted to ternary expressions and `return` statements are replaced by the expression they return.

Observation 4.6. We succeeded in implementing a flow-aware syntactic transformation, including variable inlining, which enabled us to produce a result declarative expression that preserves identifiers and has reminiscent structure of the input programs.

Readability and recognizeability of the final output is important in modernization projects since developers are expected to further evolve the generate code.

Basic metrics

The entire transformation (including grammar definitions) spans 6515 SLOC. The core C++ grammar—provided from the TXL website—has 137 nonterminal rules (595 lines of code), and we additionally (re)defined 98 grammar rules to adapt the grammar to our needs. The transformation has 171 function definitions and 297 rule definitions, making 468 definitions of operations in total.

It took 3 months full-time work of experienced software developer to implement this transformation, including learning TXL, domain understanding, unit testing and meetings with the industrial partner. The cost is deemed acceptable, especially given that the company has several similar products to modernize for which the transformation and accompanying experience would be largely reusable.

The transformation execution lasts 30 minutes on the 4119 functions out of which 105 functions are not handled; these non-handled cases were caught by our fail-fast approach, and marked by the transformation to be manually migrated.

4.4 Validating the Transformation

Our transformation uses the full expressiveness of TXL including the expressive pattern matching capabilities, implicit tree traversals, and shared global state. Furthermore, there is use of reflective features such as *dynamic reparsing* which allows serialization of abstract syntax trees to textual syntax and then reinterpretation as other syntactic structures. Individual rules and functions are written non-modularly and might intermediately break syntactic and semantic correctness properties; this makes it hard to verify individual rules compositionally, and the transformation must be validated as a whole.

Observation 4.7. Our transformation was written non-modularly, which made it impossible to do compositional verification of the transformation rules.

An concrete example of a non-modular rule in our transformation is

$$\texttt{if}\ (E)\ \texttt{return}\ \textit{true} \rightarrow \texttt{return}\ E$$

which is only correct under assumption that the `if` statement occurs last in the `main` function. Since our input size and shape is fixed, we decided it would be more feasible to treat the transformation as a black-box than try to tackle the complexity of verifying rules using white-box techniques.

Observation 4.8. We were able to treat a complex transformation as a black-box, reducing validation to checking whether the provided input and corresponding transformed output are semantically equivalent.

Approach

Ordinary TXL rules and functions are constrained to replacing well-formed syntactic trees with newly built ones, making it hard for TXL programs to produce syntactically incorrect output. The correctness criterion of the transformation was to produce *semantically* equivalent C++ programs, and therefore our validation technique must be able to reason about the semantics of C++ programs.

We considered several analysis techniques, finding that symbolic execution [King, 1976] is able to build a very precise semantic model for the programs we are considering. We decided on using the precise symbolic executor KLEE [Cadar et al., 2008] that handles the majority of features used in the code base, and integrating it in the automation process proved to be cost-effective.

One of the challenges of using KLEE is that it requires compiling the input code to LLVM [Lattner and Adve, 2004] intermediate representation (LLVM–IR), including all external libraries, because otherwise the symbolic execution may provide incorrect results. Our code base however contained calls external functions with unknown implementations, and therefore we had to instrument the code to get a closed program:

1. We created stubs for unknown functions—ordinary, static member and singleton member functions alike—such that a set of arguments is matched to the same symbolic result on every call.

2. We created stubs for the data structures with constructors that straightforwardly initialize all members to symbols and overloaded equality to do structural comparison.

Function stubs are created in the *symbolic function* style [Corin and Manzano, 2011]. For each stub, a memoization table which matches input arguments with symbolic return values is allocated. When the function is called, it looks up the arguments in the memoization table: if they are found, it returns the same symbolic result as before; otherwise, a new record that stores the arguments along with a fresh symbolic result variable is created in the table.

```
bool defined(int p) {
  static node<int, bool> *results;
  static int* counter = new int(0);
  bool* val = new bool;
  if(!getResult(&results, &results, p, counter, val)) {
    char symbolicname[40];
    sprintf(symbolicname, "defined%d", *counter);
    *val = klee_range(0, 2, symbolicname);
  }
  return *val;
}
```

Figure 4.3: A stub for an unknown Boolean function.

The stub for the Boolean function `defined` with unknown concrete implementation is presented in Figure 4.3. The static memoization table `results` and call counter are reused for all calls to the function. The function `getResults` looks up the input argument `p` in the memoization table and updating the value of the pointer `val` if `p` was already memoized. In case this is the first time `p` is encountered, then the call counter is incremented and `val` points to a new memory address, that is marked as symbolic by `klee_range`.

Observation 4.9. We were able to use off-the-shelf tools to perform semantic verification of programs, with a moderate amount of effort required to instrument the input to be useable with the tool.

Validation Experiment

Our validation experiment answers the following three questions refining RQ3:

RQ3.1 How large a part of the transformed code base can be verified automatically?

RQ3.2 How much additional effort would it require to verify the rest of the code base?

RQ3.3 How can our verification effort be generalized to other similar modernization projects?

We address RQ3.1 in Section 4.5 by running the verification procedure on the input and transformed output, reporting and classifying the results. Section 4.7 discusses the challenges (and solutions to these challenges) that appeared during verification (RQ3.2) and what parts of our verification procedure is generalizable to other transformation tools and projects (RQ3.3). Executing the validation procedure on all transformed functions lasts 7 minutes, where 3348 of the functions are evaluated trivially to pass verification–the output is identical to the input.

4.5 Bug Analysis

There were 771 functions out of 4119 in the code base that could not be trivially validated. We present the statistics in Table 4.1.

Table 4.1: Erroneous transformation cases caught by each step of the validation process.

	Step	#Cases
1	Failing transformation precondition (not handled, requiring manual inspection)	105
2	Failing silently due to unhandled syntactic structures (caught statically by TXL during preliminary steps of verification)	3
3	Caught by C++compiler	3
4	Checked for equivalence using KLEE	640
4a	Validated being equivalent	562
4b	Concrete bug cases with provided counter-examples	50
4c	False positives with spurious counter-examples (due to over-approximation of functions, and representation mismatch)	28
5	Unhandled cases containing assertions (intentional, due to design limitations of the validation technique)	20

Observation 4.10. It was possible to analyze a substantial amount of the modernized code automatically with only 20 corner cases left to be handled manually.

We analyze the bugs found by verification (3 & 4b) below. In Section 4.7 we will discuss the types of spurious counter-examples (4c), the unhandled cases due to design limitations (5), and propose solutions for these issues.

Analysis of Bugs Found by Verification

Our technique had identified seven bugs present in the transformation. While these bugs varied in nature, they had one important thing in common: they were all related to execution semantics and would have been hard to find with a syntactic check or a static semantics check using e.g., type analysis.

Observation 4.11. When the code base of our modernization project reached a certain complexity it became infeasible to find all bugs through expertise and unit testing. Validation of semantics was essential to ensure that the output code worked correctly.

Bug 4.1: ***Function call is dropped in some paths.*** The most widespread errors in the output expressions were missing function calls, that were present in the original code. This was not contained to calls of a particular function or particular place in the output expression, and other calls to the same function might still be present in the rest of the output expression.

The bug was caused by an *incomplete rewrite rule*: the rule matched a functional call in a return statement, but did not use the function call when constructing the replacement.

Bug 4.2: ***Structure replaced by a constant integer.*** Another widespread but simple bug is where the input function declares a variable with a class type, calls its object initializer with multiple arguments and then returns the variable. Here, the transformation seemed to instead return the first argument given to the object initializer which often had an incompatible type like `int`.

This bug happened due to *misuse of deep pattern search* and a *broken rule assumption*. The variable inlining procedure expected variables with simple types and so used a deep pattern match to extract the target value; for structure types, the deep pattern will however simply match the first argument value of the initializer—ignoring the rest—and use that for the inlining.

Bug 4.3: ***Conditional branches are dropped.*** This bug was also caused by *incomplete rewrite rules*, and made the transformation ignore all branches following a nested `if`-statement (also referred to in Section 4.1). The transformation rule matches a nested conditional possibly having an **else**-branch, correctly handling the nesting but not the **else**-branch.

Bug 4.4: ***The unexpected exceptions.*** This bug was surprising since it mostly happened in huge functions with heavy nesting of control flow. While the input function seems total and returns a correct result on all paths, the transformation produces an output which contains a branch that throws an exception stating that the branch should be invalid.

This bug happens due to *overconstrained pattern matching* and *broken rule assumption*. When a sequence of nested conditionals followed by a return statement is matched by the transformation, it tries to put the final return statement inside the branches of the previous conditional. However, the pattern was overconstrained—not matching the inputs it should have handled—and so the containing rule was never applied; later, when the transformation tries to convert the statement to an expression it finds a branch with no `return` statements and replaces it with a `throw` statement—as part of the fail-fast approach—since it did not expect this case to be possible.

Bug 4.5: ***Use of undeclared variables.*** This bug was caught during compilation, and happened when the original input contained declarations to local variables that were not inlined correctly by the transformation. Recall, that the transformation removes all local declarations after inling, but in this case there were still some variable accesses that were not inlined which resulted in undeclared variable errors when running the result expression by the compiler.

This bug occurs due to a combination of *dynamic reparsing capabilities* and *wrong target type* in expression. To control the number of iterations of inlining substitution, the transformation replaces variable nodes with the string representation of their assigned expressions, using the textual output capabilities of TXL. This ensures that the substitution terminates, but might also be incorrect if the transformation has not finished migrating the serialised subtrees. In general, it is caused by challenges in implementing a semantics-sensitive operation (i.e., inlining) syntactically.

Bug 4.6: ***Negation dropped in result.*** The simplest bug found by the KLEE-based verifier is where the transformation had transformed the whole input correctly except a negation operation which was missing in the output. This bug occurs due to *misuse of deep pattern search* of TXL, because a transformation rule searched for a more specific syntax type than necessary and thus ignored more complex types of terms. This bug is particularly interesting since it would not be caught by a simple type-based static analysis, but would be caught by a more sophisticated analysis such as the inductive shape analysis in Chapter 7.

Bug 4.7: ***Conditional with error code assignment dropped.*** A particularly interesting bug is where the input has a function that contains a conditional statement assigning a value to an error code pointer variable, in addition to returning a separate value. In this case, the transformation will produce output that will completely remove the conditional branch and only keep the final return value, making the function produce an incorrect result.

This bug happens due to the *dynamic reparsing capabilities* and *eager removal of source data*. It originates in the inlining phase where some abstract syntax is broken by wrongly inserted textual syntax, and subsequently a rule that removed empty conditional branches was applied.

Bug 4.8: ***Variable declarations without assignment not handled.*** This bug was caught statically in the cases when the transformation finished, but the output was empty. Similar to Bug 4.2, a combination of a *broken rule assumption* and *misuse of deep pattern search* was the cause of this bug. Inlining assumes that declaration and initialization of local variables happen in a single statement, but the cases triggering this bug had separate declaration and initialization. The declaration removal rule used deep search to identify statements containing local variable assignments and since program consisted of a single large if-statement with the assignments, it was completely removed.

Classification summary Most cases were affected by Bug 4.1 where there were 23 cases in total, and followed by Bug 4.2 which had 15 cases in total; both of which were simple in nature; this is unsurprising since function calls and object initialisations are common constructs in C++, and a simple mistake in the transformation of these features will therefore affect a large number of analyzed functions. The more interesting Bugs 4.3 and 4.4 had 5 cases in total each; this type of bugs often appeared in larger files with a complex nesting of conditionals, and would therefore have been hard to immediately spot manually or with simpler unit tests. Finally, the remaining bugs (4.5, 4.6, 4.7, 4.8) had respectively 3, 1, 1 and 3 cases in total; these errors represent corner cases that were either not caught by the preconditions of the transformation, or occurred where an intermediate assumption of the transformation was wrong.

Observation 4.12. *Simple bugs hit wide, complex bugs hit deep.* Simple semantic errors affected a large number of functions while complex errors were found in a few but bigger functions.

4.6 Formal Justification of the Procedure

The transformation translates many functions individually, each of which needs to be translated in a semantics preserving manner. We view these functions (and programs in general) as black-boxes relating input and output.

Definition 4.1. A *program P* is a set of imperative statements that state how to calculate the designated output variable ret from a set of input variables $\mathbb{X}_{\text{in}} = \{x_1, \dots, x_{\text{n}}\}$.

Definition 4.2. A *concrete store σ* is a *partial function* mapping program variables *Var* into values Val, i.e. $\sigma : \text{Var} \rightharpoonup \text{Val}$. The set of values Val range over constants from C++ base types: Boolean, bounded integer, float, etc.

A concrete execution starts with an *initial state*, σ_{in}, where all input variables are assigned some initial values. The effect of executing each statement s of program P in a state σ is a successor state σ', written as $\sigma \xrightarrow{s} \sigma'$. When all statements are executed, the program reaches a final state σ_{out}, in which the value of the output variable ret is well-defined (i.e., $\text{ret} \in \text{dom}\ \sigma$).

Definition 4.3. A *concrete execution path $\pi = \sigma_{\text{in}}, \sigma_1, \dots \sigma_{\text{out}}$* of the program P is a sequence of states, such that σ_{in} is an initial state, every next state in the sequence is obtained by sequentially executing statements from P one by one, and σ_{out} is the final state.

Definition 4.4 (Concrete program path semantics). The *path semantics* of program P—called $[\![P]\!]^{\text{trace}}$—is defined to be the set of all valid concrete execution paths π of P.

Definition 4.5 (Denotational program semantics). The *denotational semantics* of program P is a partial function $[\![P]\!] : \text{Val}^k \rightharpoonup \text{Val}$ defined by: $[\![P]\!](\sigma_{\text{in}}(i_1), \ldots, \sigma_{\text{in}}(i_k)) = \sigma_{\text{out}}(ret)$, for any concrete execution path $\pi \in [\![P]\!]^{\text{trace}} = (\sigma_{\text{in}}, \ldots, \sigma_{\text{out}})$.

Definition 4.6 (Semantic equivalence). Two programs P and P' are *semantically equivalent*, written $P \sim P'$, if for any collection of values $v_1, \ldots, v_k$ it holds: $[\![P]\!](v_1, \ldots, v_k) = [\![P']\!](v_1, \ldots, v_k)$.

Determining the semantic equivalence of programs using concrete path semantics is intractable due to the immense range of input values. Instead, we use a symbolic execution to cluster the input values using a set of constraints called path conditions.

Symbolic execution

In *symbolic execution* the program does not assign concrete values to its variables; instead, it assigns symbolic expressions containing uninterpreted symbols abstractly representing user-assignable values in a concrete execution of the program. We let Sym represent the set of uninterpretered symbols $x^?, y^?, \in$ Sym which we write as variables with a lifted question mark. In the initial execution state, each possible input variable x_i is usually assigned a corresponding unique symbol $x_i^?$.

For example, let x and y be input integer variables, then the concrete semantics of $\text{ret} = x + y$ is the set $\{([x \mapsto 0, y \mapsto 0], [x \mapsto 0, y \mapsto 0, \text{ret} \mapsto 0]), ([x \mapsto 0, y \mapsto 1], [x \mapsto 0, y \mapsto 1, \text{ret} \mapsto 1]), \ldots\}$, where initial states are all possible assignments of integer values to x and y. However, the symbolic path semantics of $\text{ret} = x + y$ will contain only one symbolic execution path $([x \mapsto x^?, y \mapsto y^?], [x \mapsto x^?, y \mapsto y^?, \text{ret} \mapsto x^? + y^?])$, where $x^?$ and $y^?$ are fresh symbols.

Symbolic execution approximates a set of different concrete paths into a single symbolic one by following all branches whenever a branching or looping statement is encountered. In the same time, for each branch it maintains a set of constraints called the *path condition*, which must hold on the execution of that path.

Definition 4.7. A *symbolic expression* $\widehat{e} \in \widehat{\text{Exp}}$ can be built out of constant values from Val, symbols from Sym, and arithmetic-logic operations.

Definition 4.8. A *symbolic store* $\widehat{\sigma}$ is a function mapping program variables Var into symbolic expressions $\widehat{\text{Exp}}$, i.e. $\widehat{\sigma} : \text{Var} \rightharpoonup \widehat{\text{Exp}}$. The initial symbolic state $\widehat{\sigma}_{\text{in}}$ maps each input variable $x_i \in \mathbb{X}_{\text{in}}$ to a fresh symbol $x_i^? \in Sym$.

Definition 4.9 (Constrained symbolic state). A *constraint* is any Boolean symbolic expression $\widehat{b} \in \widehat{\text{BExp}}$. A *constrained symbolic state* is a pair $(\widehat{\sigma}, \widehat{b})$, which constraints the symbolic expressions in $\widehat{\sigma}$ with a Boolean symbolic expression $\widehat{b}$.

Definition 4.10 (Symbolic execution path). A *symbolic execution path* of the program P is a sequence of constrained symbolic states $((\widehat{\sigma}_{\text{in}}, \text{true}), (\widehat{\sigma}_1, \widehat{b}_1), \ldots (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}))$, where the initial state $\widehat{\sigma}_{\text{in}}$ is unconstrained, and the constraint produced for the final state, $\widehat{b}_{\text{out}}$, represents the path condition.

Note how the resulting set of symbolic execution paths partitions the set of concrete execution paths. For the program that computes the absolute value of an integer variable x, there are two different paths returned by symbolic execution: $\big(([x \mapsto x^?], \text{true}), ([x \mapsto x^?, \text{ret} \mapsto x^?], x^? \geq 0)\big)$ and $\big(([x \mapsto x^?], \text{true}), ([x \mapsto x^?, \text{ret} \mapsto -x^?], x^? < 0)\big)$. If the initial value of x is non-negative, then the return value is the symbolic expression $x^?$; otherwise, the return value is $-x^?$. Hence, the set of all concrete execution paths—determined by the input values of x—has been partitioned in two sets: those for which $x \geq 0$ holds and those for which $x < 0$ holds.

Proposition 4.1. *For each concrete execution path* $\pi = (\sigma_{\text{in}}, \sigma_1, \ldots \sigma_{\text{out}})$ *of the program* P*, there exists the corresponding symbolic execution path* $\widehat{\pi} = ((\widehat{\sigma}_{\text{in}}, \text{true}), \ldots (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}))$*, such that the following equations are satisfied:*

$$\begin{aligned}
\widehat{\sigma}_{\text{in}} &= [x_i \mapsto x_1^?, \ldots, x_{\text{n}} \mapsto x_{\text{n}}^?] \\
\theta &= [x_0^? \mapsto \sigma_{\text{in}}(x_0), \ldots, x_{\text{n}}^? \mapsto \sigma_{\text{in}}(x_{\text{n}})] \\
((\widehat{\sigma}_{\text{out}})\theta)(\text{ret}) &= \sigma_{\text{out}}(\text{ret}) \\
(\widehat{b})\theta &= \text{true}
\end{aligned}$$

Theorem 4.1. *Two programs* P *and* P' *are semantically equivalent* $P \sim P'$ *iff for each value* $v \in \text{Val}$ *it holds:*

$$\bigvee_{(\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}) \in \Sigma} (\widehat{\sigma}_{\text{out}}(\text{ret}) = v \wedge \widehat{b}_{\text{out}}) \iff \bigvee_{(\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}}) \in \Sigma'} (\widehat{\sigma}'_{\text{out}}(\text{ret}) = v \wedge \widehat{b}'_{\text{out}})$$

where

$$\Sigma = \left\{ (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}) \;\middle|\; ((\widehat{\sigma}_{\text{in}}, \text{true}) \ldots (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}})) \textit{ is a symbolic path of } P \right\}$$
$$\Sigma' = \left\{ (\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}}) \;\middle|\; ((\widehat{\sigma}_{\text{in}}, \text{true}) \ldots (\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}})) \textit{ is a symbolic path of } P' \right\}$$

4.7 Experiential Reflections

Unreliability of academic tools is a know challenge in software engineering action research; due to lack of support services, these tools are rarely adopted by companies, except in rare cases where the company is able to further develop and maintain the tool in-house. Our experiences with this project tells us that this issue is less detrimental for modernization projects, since the tools are only required for use in a short transitional period and there is more freedom for customization.

The applied design and validation principles translate easily to other program and model transformation languages. Even though we verify expressive semantic properties between input and output, the transformation itself is treated as black-box and the method is oblivious to the choice of transformation language. Unclarity remain regarding the generalizability of the technique with regards to transformations that should support a broader variety of input, and whether the identified bugs are specific to this case, this input and output languages, and TXL.

Validation Challenges

In practical projects, many of the idealized assumptions made in research are not fulfilled, e.g., that data is represented in a particular way or that we have access to all library code at validation time. We will discuss three concrete issues we had when validating the transformed code, and how we effectively mitigated them.

Representation of Boolean expressions

In C++ integer valued expression can be used as logical tests (inside `if`-statements etc.), where any non-zero value counts as true and zero counts as false. If an integer variable a is used only as a logical condition both in the input and output programs it would be pragmatically fine. However, our transformation contains simplification rules which convert statements of form `if (a) return true; else return false;` to `return a;` which clearly has different semantics for non-Boolean values. In cases where we are certain that specific integer variables are only used as conditionals we instruct KLEE to assume that these variables have values lying in range $[0, 2)$.

Over-approximation of Function Semantics

We use stubs to model external functions whose implementation we do not know, which is over-approximating since we only know that equal arguments provide the same result values. If any of these functions however had equivalent implementations and we used a different stub for each one, calling the two stubs with the same parameters would result in distinct return values. This led to a number of false positives where KLEE decided that the input and output programs were not equivalent, and was solved by using the same stub for functions which were known to be identical *a priori*.

Assertions

When KLEE meets a C++ assertion that fails on a possible path, it will immediately halt execution for that particular path. This concretely means that it will never check whether the input and output functions have the same results, or in this case rather both fail. Instead of using the default assertion function, one could instead use a stub that throws a recoverable error on failing conditions; thereafter, one could check whether both the input and output programs failed on the same paths and if they did one could consider the paths to be equal. Our code base contains 20 cases affected by this limitation, but implementing the suggested solution was not feasible in the allocated time.

4.8 Related Modernization Efforts

Semantic Designs (SD)[5] presents a similar automation-based modernization effort is presented in a joint project with Boeing[6] [Akers et al., 2007] which uses Semantic Designs's commercial transformation and analysis tool DMS [Baxter et al., 2004] to convert an old component-based C++ code base to a standardized CORBA [Siegel, 2000] architecture. This is in contrast to our case study which relied solely on freely

[5]`https://www.semanticdesigns.com`
[6]`http://www.boeing.com`

available tools like TXL and KLEE, and we more importantly had a significantly more thorough validation effort with precise descriptions of the challenges we encountered and the lessons we learned.

A series of papers [Selim et al., 2013, 2014, 2015] discuss a case study that aims to design and verify a model transformation for modernizing an existing collection of proprietary models to conform to the standardised AUTOSAR [Bunzel, 2011] format. The transformation [Selim et al., 2015] was initially encoded in a limited expressive subset of the ATL model transformation language [Jouault et al., 2008] and then verified for structural properties [Selim et al., 2013]. The same verification effort was then repeated [Selim et al., 2014] more efficiently by by symbolically executing a version of the transformation re-encoded in DSLTrans [Barroca et al., 2010]. While these verification tools and the presented case study have significant contributions to the model transformations community, they were not applicable in our study due to the difference in expressiveness between TXL and the verified non-Turing-complete subset of ATL/DSLTrans, and the complexity of the property we wanted to check (behavioral equivalence versus structural properties).

4.9 Recap

This chapter presented a collaborative effort focusing on designing and verifying an industrial modernization project implemented using the high-level transformation language TXL. It showed the usefulness of the high-level transformation language features—such as traversals and fixed-point iteration in the rule-based execution—in providing a practical and cost-effective development cycle for migrating a large code base. More importantly for this dissertation, the modernization project highlighted the practical need for formal verification methods such as symbolic execution: even though our the modernization transformation was written by an expert transformation programmer and had been well-tested, the validation effort found seven serious bugs spread over 50 concrete instances. We furthermore provided a detailed analysis of the bugs and their causes, which was a particularly useful inspiration for directing the verification work in the rest of the thesis.

Chapter 5

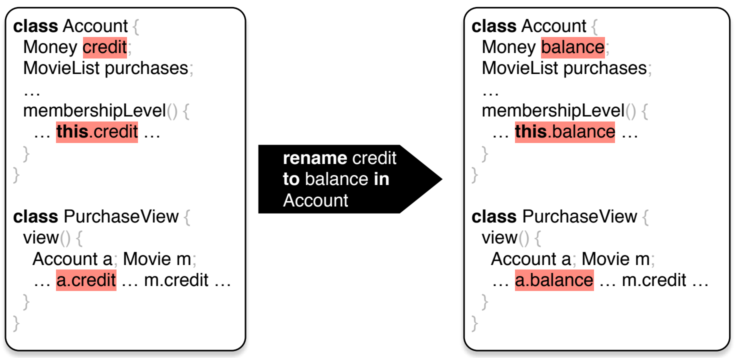
Symbolic Execution of High-Level Transformation Languages[*]

The *rename field* refactoring in Figure 5.1 is an example of a transformation that is best written in a high-level transformation language. The transformation changes the name of a given field in the definition of a class and ensures that all relevant field accesses use the new field name [Fowler, 1999].

This transformation, while simple in nature, requires a lot of boilerplate to define in traditional programming languages—like C or Java—since one needs to construct visitors that recursively traverse each of the many cases in the target program syntax tree. In contrast, high-level transformation languages provides these features directly in the form of *type-directed querying and manipulation*. These operations allow *deep* matching and rewriting of structures by following the types of objects and references between them. In our example in Figure 5.1, a type-directed query makes it possible to retrieve all accesses to the `credit` field where the target expression has type `Account`, using only a couple of lines.

Transformations are complex programs and as such prone to bugs; for our *rename field* example, a bug could be that we modified accesses to fields with similar names but an unrelated class. Due to the complexity of transformations these bugs are hard to find and expensive to fix; this creates a need for automated formal techniques that support verifying the correctness of transformations, which is an important step to-

[*]This chapter is based on Al-Sibahi et al. [2016] which is the technical report version of the conference paper, Al-Sibahi et al. [2016].



```
class Account {
  Money credit;
  MovieList purchases;
  …
  membershipLevel() {
    … this.credit …
  }
}

class PurchaseView {
  view() {
    Account a; Movie m;
    … a.credit … m.credit …
  }
}
```

```
class Account {
  Money balance;
  MovieList purchases;
  …
  membershipLevel() {
    … this.balance …
  }
}

class PurchaseView {
  view() {
    Account a; Movie m;
    … a.balance … m.credit …
  }
}
```

Figure 5.1: An example execution of the *Rename-Field* refactoring: rename the definition of credit to balance and update all references accordingly.

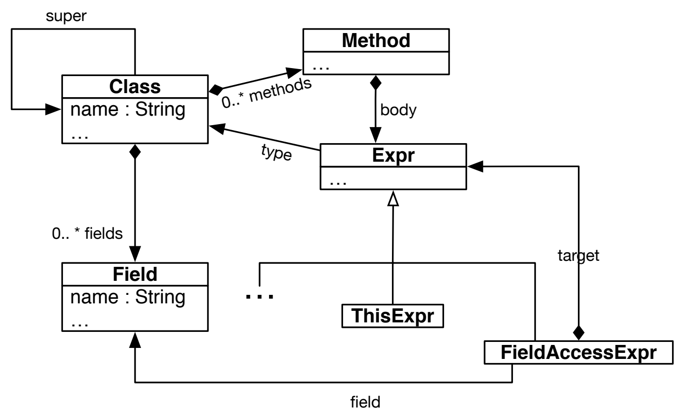
wards increasing the trustworthiness of our language implementations and tools [Cadar and Donaldson, 2016; Schäfer et al., 2009; Hoare, 2005].

In this chapter, we present a foundational symbolic execution technique that handles high-level transformation features as first-class. Concretely, our contributions are:

- A formal symbolic execution technique supporting complex concepts such as symbolic sets, ownership constraints, type-directed querying and manipulation, and fixed-point iteration.
- An evaluation of the symbolic execution technique when used for white-box test generation using realistic model transformations and refactorings, showing that our symbolic executor makes white-box test generation for transformation feasible.
- A comparison of our symbolic execution technique to object-oriented symbolic executors, highlighting the difficulties of dealing with target high-level features as second-class.

5.1 Overview

Let us start by discussing our motivating *rename field* example (Figure 5.1) in more detail. Observe that the refactoring program needs two key components: *type definitions* (sometimes called *meta-model*) for the data and the actual transformation *code*. We will use (minimalistic)



(a) Abstract syntax for simple object-oriented programs

```
input:
  target_class: Class,
  old_field: Field,
  new_field: Field
precondition:
  old_field ∈ target_class.fields
    ∧ new_field ∉ target_class.fields

target_class.fields :=
  (target_class.fields \ old_field) ∪ new_field
foreach faexpr ∈ target_class match* FieldAccessExpr do
  if faexpr.field = old_field ∧
      faexpr.target.type = target_class then
        faexpr.field := new_field
  else skip
```

(b) The *Rename-Field* Refactoring implemented in the small formal transformation language, TRON

Figure 5.2: A simplified version of the *rename-field* refactoring example in TRON

class diagrams to show the former (e.g., Figure 5.2a), and a compact formally defined transformation language TRON for the latter (e.g., Figure 5.2b). These two notational choices incorporate some key common characteristics of transformations, which we discuss below.

Figure 5.2a shows the types for the abstract syntax of a hypothetical object-oriented language, which we will use to model the programs we are refactoring. We show the classes[1] and properties relevant for our

[1]The abstract syntax types describe a hypothetical object-oriented programming language and not of the formal core transformation language (TRON).

refactoring, while omitting irrelevant details. In our example, each class has a name (an attribute), and *contains* two collections, one for fields and one for methods. Recall that in class diagrams, a black diamond is used to represent *containment* references, which are traditionally found both in object-oriented modeling languages and grammar-based languages like TXL.

Additionally, each class may simply *refer* to a possible super-class, which is denoted using a line without the diamond symbol. All the represented features—classes, containment references, simple references, and simply-typed attributes—are typical of transformation languages we want to handle by our technique.

For simplicity of presentation, we let the body of each method in our example be an expression. Expressions themselves can come in many different kinds (thus the use of *inheritance*), but we only show expressions representing **this** and field access expressions since they are the ones relevant for the example.

A simplified implementation of the *rename field* refactoring, is presented in Figure 5.2b using the core transformation language (TRON). We will discuss the example based on general intuitions, presenting the formal semantics of the core language in Section 5.2–Section 5.3. In the start (Lines 1–4), we list the input parameters—references to a class, the field with the old name, and the replacing field with the new name—and the application precondition (Lines 5–7) which specifies that the old field has to be contained in the fields of the input class, whereas the new must not.

We begin the refactoring by removing the old field definition from the fields of the class and adding the new field definition (Lines 9–10). Then, in Line 11, all field access expressions in the class are matched and gathered into a single set using a *deep type-directed query*, which collects instances of FieldAccessExpr contained transitively in the input class. After the deep type-directed query, Line 11 binds each element of the matched objects to *faexpr* executing Lines 12–15 for each of these objects. If the expression accesses the refactored field (Lines 12–13) then the field reference is updated to point at the new field (Line 14). It is typical for transformation languages that references are redirected or attributes are changed.

The example demonstrates how TRON, despite being minimalistic, supports the key high-level transformation features such as type-directed querying and iteration via **foreach**, collection operations, and imperative updates for manipulation (like ATL or Rascal).

Symbolic Execution of Transformations

Symbolic execution is an effective way to check the presence of bugs in transformations, since it systematically explores the various transformation program paths. Figure 5.3 presents an overview of our symbolic execution technique for transformations; here, the symbolic executor expects as input the transformation program, along with the required type definitions. The initial step is to run the symbolic executor (see Section 5.3) on the input transformation and generate a finite set of path conditions. These path conditions are logical formulae constraining the shape, types and range of input data, achieved by refining input constraints according to the semantics of each statement in the given transformation.

We use the model finder to prune those paths which produce unsatisfiable formulae so that only valid paths are considered. In our implementation, the model finder uses the relational constraint solver KodKod [Torlak and Jackson, 2007] to check the existence of a suitable model satisfying a target formula within a bounded scope, possibly failing when either the formula is unsatisfiable or the scope is too small.

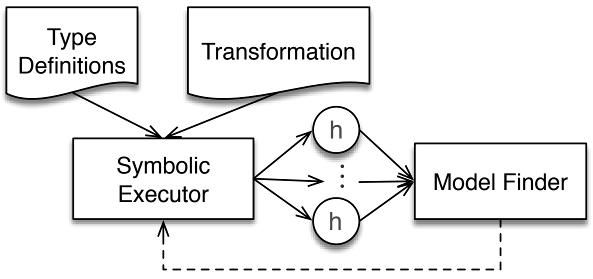


Figure 5.3: High level architecture of the symbolic executor.

5.2 A Demonstration Language

We developed the small formal transformation language TRON as a methodological device, to keep the formal work, discussions, and the presentation focused, and to allow agile experimentation; TRON is a decoy language*not* meant to be used by programmers, but is designed so that our ideas remain applicable to real-world transformation languages.

Table 5.1 shows how TRON incorporates characteristic features of high-level transformation languages. We present TRON in two parts: i) the meta-model that captures structures of the manipulated data, and ii) the operational part of the language that describes computations.

Language / TRON Feature	ATL	Haskell	Maude
Containment	Containment references	Algebraic Data Types	Many-Sorted Terms
Set expressions	OCL collections and collection operations	Standard library	Standard library
Shallow matching	Type testing via `oclIsKindOf`	Pattern matching	Rewrite rules
Deep matching	Transformation rule definition	Generic traversal via Uniplate	Rewrite rules and strategies
Fixedpoint iteration	Lazy rules, recursive helpers	Recursive functions	Rewrite strategies

Table 5.1: Relating TRON features to existing high-level transformation languages

Data Model. The data in TRON is described by types that capture the common features of high-level transformation languages: constructors, containment, references and generalization. It is essentially a formal model for the kind of structures like the one represented in Figure 5.2a.

A data model is a tuple: $(\text{Class}, \text{Field}, \mathsf{gen}, \mathsf{ref})$, where Class is the set of *classes*, Field is the set of *fields*, partitioned into *contained* fields $\text{Field}_{\blacklozenge}$ and *referenced* fields $\text{Field}_{\rightsquigarrow}$. Later, we use c to range over class names (Class), and f to range over field names (Field). A class has at most one superclass, described by the generalization relation: $\mathsf{gen} \subseteq \text{Class} \times \text{Class}$, where $c \;\mathsf{gen}\; c'$ means that c is a subtype of c'. Each field has a corresponding type, a class. This is represented by the references relation $\text{ref} \subseteq \text{Class} \times \text{Field} \times \text{Class}$, where $\mathsf{ref}(c, f, c')$ means that the class c has a field f of type c'. We generally expect that gen has the expected properties of a generalization relation, namely that there is a strict ordering of generalization (no cycles); similarly, we expect that reference definitions in ref are not overriden by subtypes, i.e. if for any class c a supertype has defined a typing for a field f, then c must have the same typing for f.

To get all the fields defined for a class c or any of its supertypes, we define the following function:

$$\mathsf{fields}(c) = \left\{(f, c'') \;\middle|\; c \;\mathsf{gen}^*\; c' \land \mathsf{ref}(c', f, c'')\right\}$$

We do not explicitly handle simple types in our formal model, instead they can be modeled theoretically as classes. For instance, we assume a class Integer with instances representing integer numbers; integer attributes can then be modeled as references to this class. We do handle simple types in our symbolic execution tool by using symbolic variables of corresponding simple types.

Heap Representation. Concrete TRON programs are executed over finite concrete heaps ($h \in \text{Instance} \times \text{Field} \rightarrow \wp(\text{Instance})$) that contain instances organized into structures using containment links and simple links (a link is a concrete instantiation of a field). In particular each link f of an instance o, can point to a set of instances os. Instances are typed at runtime using a type environment ($\Gamma \in \text{Instance} \rightarrow \text{Class}$). An example heap is presented in Figure 5.4, which describes a possible definition of the Account class used in Figure 5.1.

For the remainder of this chapter we will only consider well-formed heaps where all instances are typed and their structure conforms to the static typing provided by the data model. Furthermore, we assume that in well-formed heaps each instance can at most be pointed to by a single containment link (no sharing) and that there are no cycles in containment (acyclicity). Note that these restrictions do not apply for simple links, which still allow cycles and sharing.

Abstract Syntax. The core TRON constructs include access to variables and fields, constants, object construction, assignment, sequencing and

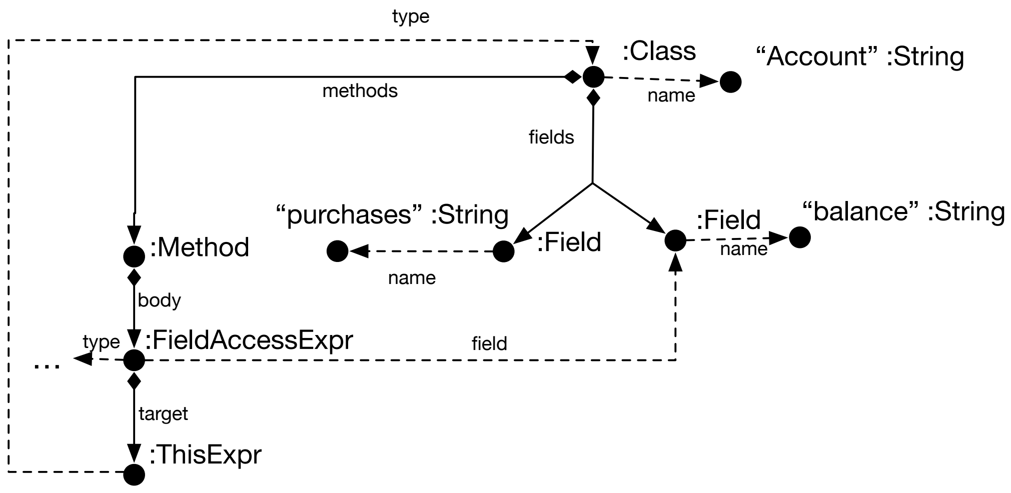


Figure 5.4: A heap instantiating one model of Figure 5.2a, inspired by the Account class in Figure 5.1. Dots (•) represent instances, diamond affixed lines represent containment links, dashed lines represent simple links.

branching. The syntax is summarized in the following grammar:

$$\begin{aligned}
\text{SetExpr} \ni e &::= x \mid \emptyset \mid e_1 \cup e_2 \mid e_1 \cap e_2 \mid e_1 \setminus e_2 \\
\text{BoolExpr} \ni b &::= e_1 \subseteq e_2 \mid e_1 = e_2 \mid \neg b \mid b_1 \wedge b_2 \\
\text{MatchExpr} \ni me &::= e \mid e \ \textbf{match} \ c \mid e \ \textbf{match*} \ c \\
\text{Statement} \ni s &::= \textbf{skip} \mid s_1; s_2 \mid x := e \mid x := e.f \\
&\quad \mid x := \textbf{new} \ c \mid e_1.f := e_2 \mid \textbf{if} \ b \ \textbf{then} \ s_1 \textbf{else} \ s_2 \\
&\quad \mid \textbf{foreach} \ x \ \textbf{in} \ me \ \textbf{do} \ s \mid \textbf{fix} \ e \ \textbf{do} \ s
\end{aligned}$$

where x is a variable, f is a field name, and c is a class name. The set expressions (e) and Boolean expressions (b) are standard. Match expressions (me) include "e **match** c" which allows finding all objects computed by e that are instances of class c. For example, consider a set of objects representing program expressions to be referenced $exprs = \{te_1, te_2, fae_1, fae_2\}$ where te_i is of type ThisExpr (Figure 5.2a) and fae_i is of type FieldAccessExpr. The TRON expression "$exprs$ **match** ThisExpr" returns the set $\{te_1, te_2\}$, similarly "$exprs$ **match** FieldAccessExpr" returns $\{fae_1, fae_2\}$ and "$exprs$ **match** Expr" return the complete set $exprs$.

A deep variant of the pattern matching, e **match*** c, is also provided. It matches objects nested at an arbitrary depth inside other objects, following the containment references ($\text{ref}_{\blacklozenge}$). This is similar to the matching capabilities in many of the model transformation, term and graph rewriting languages. A classical example here would be to get all objects representing program variables in a term, i.e., the expression *expr* **match*** Var—for a class Var representing variables—would return a set that has all variables transitively contained in *expr*.

Most of the statements, s, are standard formulations from Java or IMP; from left to right, the statements are: skip, sequencing, branching, variable assignment, assignment of a field value, object creation (**new**) and assignment to a field.

There are two looping constructs in TRON. The "**foreach** x **in** *me* **do** s" iterates over the set of elements matched by *me*, binding each element to x, and executes then statement s for each of them. The "**fix** e **do** s" loop executes the body s, and continues to do so as long as the values of e after and before iteration differ; therefore expression e defines the part of the heap which is relevant for this fixed point iteration (a control condition). By allowing the statement to explicitly depend on a local control condition, it is possible to create temporary helper values on the heap (outside e) without influencing the loop termination. This allows explicit modeling of the implicit fix point iteration that is also supported by many high-level transformation languages where rewrite rules are repeatedly applied until no rule is further applicable.

Concrete Semantics

The formal concrete semantics of the TRON constructs is presented in Figure 5.5 and Figure 5.6. Most constructs are handled as expected from the informal description: the only constructs that require more elaboration are match expressions and field updates. Match expressions evaluation requires the auxiliary functions match, which gathers the subset of instances which are of a subtype of the target type, and the auxiliary function dcs, which collects all are transitively contained instances for a given set of instances in the heap. Field updates require that assigned values are correctly typed via the typed relation, and ensure to remove old containment references to the assigned instances in case of assignment to a containment reference (change of ownership).

Set Expressions

$$\mathcal{E}[\![e]\!]\sigma \in \wp(\text{Instance})$$
$$\mathcal{E}[\![x]\!]\sigma = \sigma(x) \quad \mathcal{E}[\![\emptyset]\!]\sigma = \emptyset$$
$$\mathcal{E}[\![e_1 \cup e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma \cup \mathcal{E}[\![e_2]\!]\sigma$$
$$\mathcal{E}[\![e_1 \cap e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma \cap \mathcal{E}[\![e_2]\!]\sigma$$
$$\mathcal{E}[\![e_1 \setminus e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma \setminus \mathcal{E}[\![e_2]\!]\sigma$$

Boolean Expressions

$$\mathcal{B}[\![b]\!]\sigma \in \{\text{tt}, \text{ff}\}$$
$$\mathcal{B}[\![\neg b]\!]\sigma = \neg\mathcal{B}[\![b]\!]\sigma$$
$$\mathcal{B}[\![e_1 \subseteq e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma \subseteq \mathcal{E}[\![e_2]\!]\sigma$$
$$\mathcal{B}[\![e_1 = e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma = \mathcal{E}[\![e_2]\!]\sigma$$
$$\mathcal{B}[\![b_1 \wedge b_2]\!]\sigma = \mathcal{B}[\![b_1]\!]\sigma \wedge \mathcal{B}[\![b_2]\!]\sigma$$

Match Expressions

$$\mathcal{M}[\![me]\!](\sigma, \Gamma, h) \in \wp(\text{Instance})$$
$$\mathcal{M}[\![e]\!](\sigma, \Gamma, h) = \mathcal{E}[\![e]\!]\sigma$$
$$\mathcal{M}[\![e \text{ match } c]\!](\sigma, \Gamma, h) = \text{match}(\mathcal{E}[\![e]\!]\sigma, c, \Gamma)$$
$$\mathcal{M}[\![e \text{ match}^* \ c]\!](\sigma, \Gamma, h) = \text{match}(\text{dcs}(\mathcal{E}[\![e]\!]\sigma, h), c, \Gamma)$$

$$\text{match}(os, c, \Gamma) = \{o | o \in os \wedge \Gamma(o) \text{ gen}^* \ c\}$$

$$\text{dcs}(os, h) = \left\{o' \middle| o \in os \wedge o \blacklozenge\!\!\rightarrow_h^* o'\right\}$$

$$o \blacklozenge\!\!\rightarrow_h o' \text{ iff } \exists f \in \text{Field}_\blacklozenge . o' \in h(o, f)$$

Figure 5.5: Concrete semantics for TRON

Statements

$$\text{update}(o,f,os,h) = \begin{cases} h[(o,f) \mapsto os] & \textbf{if } f \in \text{Field}_{\leadsto} \\ h'[(o,f) \mapsto os] & \textbf{if } \begin{array}{c} f \in \text{Field}_{\blacklozenge} \wedge \\ \nexists o' \in os. o' \ \blacklozenge\!\!\to_h^* \ o \end{array} \end{cases}$$

$$\text{where } h' = [(o,f) \mapsto \text{do-f}(h(o,f),f,os) \,|\, (o,f) \in \textbf{dom}\, h]$$

$$\text{do-f}(os',f,os) = \begin{cases} os' & \textbf{if} f \in \text{Field}_{\leadsto} \\ os' \setminus os & \textbf{if} f \in \text{Field}_{\blacklozenge} \end{cases}$$

$$\text{typed}_{\Gamma}(o,f,os) \text{ iff } \exists c.(f,c) \in \Gamma(o) \wedge \forall o' \in os. \Gamma(o') \ \mathsf{gen}^* \ c$$

$$\frac{}{\textbf{skip}, \sigma, \Gamma, h \Longrightarrow \sigma, \Gamma, h} \qquad \frac{\begin{array}{c} s_1, \sigma, \Gamma, h \Longrightarrow \sigma'', \Gamma'', h'' \\ s_2, \sigma'', \Gamma'', h'' \Longrightarrow \sigma', \Gamma', h' \end{array}}{s_1; s_2, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}$$

$$\frac{\sigma' = \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]}{x := e, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma, h} \qquad \frac{\begin{array}{c} o' \text{ fresh} \\ \sigma' = \sigma[x \mapsto \{o'\}] \qquad \Gamma' = \Gamma[o' \mapsto c] \\ h' = h\,[(o',f) \mapsto \emptyset | (f,-) \in \mathsf{fields}(c)] \end{array}}{x := \textbf{new}\ c, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}$$

$$\frac{\mathcal{E}[\![e]\!]\sigma = \{o\} \qquad \sigma' = \sigma[x \mapsto h(o,f)]}{x := e.f, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma, h} \qquad \frac{\begin{array}{cc} \mathcal{E}[\![e_1]\!]\sigma = \{o\} & \mathcal{E}[\![e_2]\!]\sigma = os \\ \text{typed}_{\Gamma}(o,f,os) & \text{update}(o,f,os,h) = h' \end{array}}{e_1.f := e_2, \sigma, \Gamma, h \Longrightarrow \sigma, \Gamma, h'}$$

$$\frac{\mathcal{B}[\![b]\!]\sigma = \text{tt} \qquad s_1, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}{\textbf{if}\, b\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'} \qquad \frac{\mathcal{B}[\![b]\!]\sigma = \text{ff} \qquad s_2, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}{\textbf{if}\, b\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}$$

$$\frac{\begin{array}{c} s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h' \\ \mathcal{E}[\![e]\!]\sigma = \mathcal{E}[\![e]\!]\sigma' \end{array}}{\textbf{fix}\, e\ \textbf{do}\ s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'} \qquad \frac{\begin{array}{c} s, \sigma, \Gamma, h \Longrightarrow \sigma'', \Gamma'', h'' \\ \mathcal{E}[\![e]\!]\sigma \neq \mathcal{E}[\![e]\!]\sigma'' \\ \textbf{fix}\, e\ \textbf{do}\ s, \sigma'', \Gamma'', h'' \Longrightarrow \sigma', \Gamma', h' \end{array}}{\textbf{fix}\, e\ \textbf{do}\ s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}$$

$$\frac{\mathcal{M}[\![me]\!](\sigma, \Gamma, h) = os \qquad x \mapsfrom os \vdash s, \sigma, \Gamma, h \overset{\text{each}}{\Longrightarrow} \sigma', \Gamma', h'}{\textbf{foreach}\ x \in me\ \textbf{do}\ s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'}$$

$$\frac{}{x \mapsfrom \emptyset \vdash s, \sigma, \Gamma, h \overset{\text{each}}{\Longrightarrow} \sigma, \Gamma, h} \qquad \frac{\begin{array}{c} s, \sigma[x \mapsto \{o\}], \Gamma, h \Longrightarrow \sigma'', \Gamma'', h'' \\ x \mapsfrom os \vdash s, \sigma'', \Gamma'', h'' \overset{\text{each}}{\Longrightarrow} \sigma', \Gamma', h' \end{array}}{x \mapsfrom \{o\} \uplus os \vdash s, \sigma, \Gamma, h \overset{\text{each}}{\Longrightarrow} \sigma', \Gamma', h'}$$

Figure 5.6: Concrete semantics for TRON (Cont.)

5.3 Symbolic Execution

We discuss the main design principles of our symbolic executor. Although, the technique has been developed for TRON, the design decisions were driven by the desire to handle the modeled high-level transformation language in general.

Representing Rich States Symbolically

Spatial Constraints. Transformations manipulate structured data, not just simple values, and so the symbolic states of our executor describe primarily the possible shapes of the memory heap. Following other symbolic executors for object-oriented languages [Khurshid et al., 2003], we use *spatial constraints* to restrict the shapes admitted by an execution path. These constraints are first order formulae restricting values that are pointed to by links. In the style of the Lazier# algorithm [Deng et al., 2012], we distinguish between two kinds of symbolic objects: *symbolic instances* and *symbolic references*.

A *symbolic instance* ($\widehat{o} \in \widehat{\text{Instance}}$) abstracts over a unique instance. Instances cannot alias, so two different symbolic instances always point to two different class instances in memory, even if they have the same type. A *symbolic reference* ($x^?, y^? \in \text{Symbol}$) points to a class instance that may be aliased by other reference symbols, and, indeed, by some symbolic instances. The separation of symbolic instances and symbolic references allows separating reasoning about the structural representation of data from aliasing by references. We can lazily reason about aliasing without committing pre-maturely to a particular concretization of the heap structure. This is particularly important for our symbolic executor, as it handles deep containment constraints, which are hard to reason about and are heavily affected by aliasing (more about deep containment constraints below).

In traditional symbolic execution [Khurshid et al., 2003], whenever a field is accessed, the executor branches to initialize it to a new symbolic instance, or to alias an existing symbolic instance. In contrast, the Lazier# algorithm, simply assigns a distinct symbolic reference to each fresh field access and aliasing is only explicitly treated if the substructure of that symbolic reference is further explored.

For objects created using **new**, we eagerly generate a new concrete instance and exclude it from aliasing with pre-existing symbolic references, as new objects cannot alias previously existing ones (assuming correctness of the memory manager). To emphasize this in the rules below, we mark the explicitly created instances with a dagger ($\widehat{o}^{\dagger}$).

Set Symbols. In addition to ordinary symbolic references, we introduce *set symbols*, in the style of Cox et al. [2015] ($X^?, Y^? \in \text{SetSymbol}$), which abstract over finite sets of instances with unknown cardinality. This addition may seem very simple at first, but is key for our symbolic executor: it allows us to range over sets without prematurely concretizing their cardinality, contained objects, or their structure and aliasing.

Set Expressions and Set Constraints. Set symbols can be combined using symbolic set expressions:

$$\widehat{\text{SetExpr}} \ni \widehat{e} \ ::= \ X^? \mid \emptyset \mid \{x_0^?, \ldots, x_\text{n}^?\} \mid \widehat{e}_1 \cup \widehat{e}_2 \mid \widehat{e}_1 \cap \widehat{e}_2 \mid \widehat{e}_1 \setminus \widehat{e}_2$$

The symbolic set expressions mimic the set expressions of TRON, presented in Section 5.2, but without match expressions and with support for literal set constructors over simple symbolic references $\{x_1^?, \ldots, x_\text{n}^?\}$. The meaning of the latter is a set of a fixed cardinality n, whose all elements are distinct (so, as a side effect, it also precludes aliasing between symbols listed). We use it to concretize the cardinality and content of sets during iteration.

Set expressions are embedded into constraints in a standard manner, using subset and equality constraints:

$$\widehat{\text{BoolExpr}} \ni \widehat{b} \ ::= \widehat{e}_1 \subseteq \widehat{e}_2 \mid \widehat{e}_1 = \widehat{e}_2 \mid \neg\widehat{b} \mid \widehat{b}_1 \wedge \widehat{b}_2$$

During symbolic execution the set comprehensions and reference symbols interplay to our benefit, allowing to describe assumptions about sets more lazily. For example, consider the constraint that equates two sets of cardinality 3 of unknown references: $\{x_1^?, x_2^?, x_3^?\} = \{y_1^?, y_2^?, y_3^?\}$. Generating this constraint allows to avoid deciding prematurely, which of the six possible aliasing configurations between x_is and y_js we are seeing, something which would not scale if done repeatedly.

Containment Constraints. A special feature of our symbolic executor is its ability to reason about the deep containment constraints of the manipulated data structures, which are extremely common in language processing (abstract syntax trees) and in data modeling. Besides eliminating many false positives, reasoning about containment also allows implementing deep matching.

To model deep containment constraints, we define a containment relation as the union of all links typed by containment fields, and insist that, for any two objects, their containments sets (the transitive closure of the containment relation) are disjoint. Furthermore, we enforce the acyclicity of the containment relation, ensuring that the irreflexive transitive closure of containment does not contain the identity pair for any object. We are using a solver (KodKod) that allows reasoning about transitive closures.

In order to perform symbolic deep matching, we on-demand bind a symbolic *containment set* for each instance $\widehat{o}$ used in a deep-matching query against a type c, to keep track of descendants. For example, this allows us to track all instances representing program variables for a some symbolic instance $\widehat{o}$ representing a program expression. This set is further constrained and unfolded during execution, in order to maintain sound access to instances that are contained by $\widehat{o}$.

Type Constraints. We introduce type approximation in our symbolic executor, in order to not concretize types of instances (objects) prematurely. Many transformation rules operate on data constrained by types with inheritance, so the actual type of parameters might be unknown during symbolic execution. We maintain a bounding constraint on types, and refine it during execution by-need during branching and concretization cycles.

A type constraint environment ($\widehat{\Gamma}$) maps each symbolic reference, set symbol, and each, symbolic instance $\widehat{o}$ to a type bound $(cs_{\text{in}}, cs_{\text{ex}})$. The bound restricts the types of concrete values assignable to a symbol in question. A type bound is a tuple $(cs_{\text{in}}, cs_{\text{ex}})$ where the first component cs_{in} is a set of classes that specify the possible supertypes of a symbol and the second component cs_{ex} is a set of classes that specify excluded supertypes of a symbol, e.g. $(\{\text{Expr}\}, \{\text{ThisExpr}, \text{FieldAccessExpr}\})$ represents all subtypes of Expr (expressions) that are not subtypes of ThisExpr or FieldAccessExpr ('this' and field access expressions).

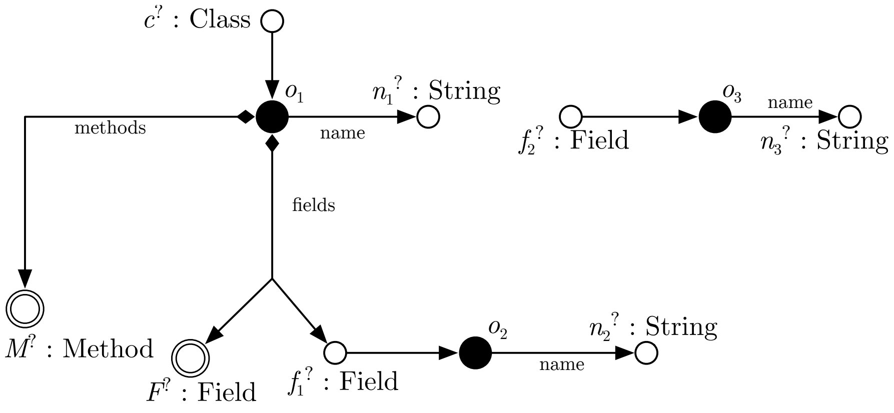
We only consider type bounds $(cs_{\text{in}}, cs_{\text{ex}})$ that are *well-formed*. That is: (i) the set of possible super-types cs_{in} cannot be empty, and (ii) none of the super-types is excluded: there is no class $c \in cs_{\text{in}}$ which is a subtype of an excluded super-type $c' \in cs_{\text{ex}}$. We also maintain an invariant that the set of possible super-types cs_{in} given by Γ is a singleton for symbolic instances. We will simply write c as a shorthand for $(\{c\}, \{c' \mid c' \text{ gen } c\})$ when the type of an element is precisely known.

Symbolic Heaps. A *symbolic heap* combines all types of constraints discussed above to describe possible concrete heaps and typings that could have been created during the execution. We define a symbolic heap to be a tuple $(\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b})$, where $\widehat{z} \in \text{Symbol} \rightarrow \widehat{\text{Instance}}$ is a symbolic reference environment—partial mapping of symbolic references to symbolic instances that they are constrained to point to; $\widehat{\ell} \in \widehat{\text{Instance}} \times \text{Field} \rightarrow \widehat{\text{SetExpr}}$ collects the symbolic instances, by mapping fields of symbolic instances to symbolic set expressions; $\widehat{d}$ is an environment storing deep containment constraints $\widehat{d} \in \widehat{\text{Instance}} \times \text{Class} \rightarrow \widehat{\text{SetExpr}}$ for all symbolic instances, Γ is the type constraint environment, and b is the path constraint so far, in the execution leading to this symbolic heap.

An example symbolic heap is presented in Figure 5.7 using both the above syntax and a diagram[2]. Dot vertices (•) denote symbolic instances, white circles (○) denote symbolic references and large double-stroked white circles (◎) denote symbolic reference sets.

Satisfiability of Symbolic Heaps We say that $\widehat{h}$ is *satisfiable* if there is at least a pair of a concrete heap h and type environment Γ that are consistent with the constraints present in $\widehat{h}$. To check the consistency of the concrete heap h and type environment Γ against the constraints in $\widehat{h}$ ($h, \Gamma \models^{m} \widehat{h}$) we need a model $m \in (\text{Symbol} \rightarrow \text{Instance}) \cup (\text{SetSymbol} \rightarrow \wp(\text{Instance}))$ which assigns to each symbolic reference a concrete instance, and to each symbolic reference set a concrete set of instances; we assume that symbolic instances are mapped directly one-to-one to concrete instances. The symbolic reference environment $\widehat{z}$ is satisfied by model m, if m is an extension of $\widehat{z}$, i.e., $\forall x \in \operatorname{dom} \widehat{z}. \widehat{z}(x) = m(x)$. The symbolic shape environment $\widehat{\ell}$ is consistent with heap h, if they agree on the structure of all defined links given the model m, i.e. $\forall (o, f) \in \operatorname{dom} \widehat{\ell}. m(\widehat{\ell}(o, f)) = h(o, f)$; here the application of m to set expressions is extended to work by replacing all sub symbolic references and symbolic reference sets in the set expression with their value in m. The heap h and type environment Γ are consistent with the deep containment constraints $\widehat{d}$ if $\widehat{d}$ capture all necessary descendants for each class c for a particular instance

[2]We use our own diagram notation for objects instead of the UML one because it is more compact and allows us to neatly represent non-standard concepts like symbolic values and containment.



Symbolic references $\widehat{z} = [c^? \mapsto \widehat{o_1}, f_1^? \mapsto \widehat{o_2}, f_2^? \mapsto \widehat{o_3}]$

Symbolic instances $\widehat{\ell} = [\,(\widehat{o_1}, \text{name}) \mapsto n_1^?, (\widehat{o_1}, \text{methods}) \mapsto M^?, (\widehat{o_1}, \text{fields}) \mapsto F^? \uplus \{f_1^?\}, (\widehat{o_2}, \text{name}) \mapsto n_2^?, (\widehat{o_3}, \text{name}) \mapsto n_3^?]$

Containment constraints $\widehat{d} = []$ (not accumulated yet)

Type constraints $\widehat{\Gamma} = [\widehat{o_1} \mapsto \text{Class}, \widehat{o_2} \mapsto \text{Field}, \widehat{o_3} \mapsto \text{Field}, c^? \mapsto \text{Class}, n_1^? \mapsto \text{String}, f_1^? \mapsto \text{Field}, n_2^? \mapsto \text{String}, f_2^? \mapsto \text{Field}, n_3^? \mapsto \text{String}, M^? \mapsto \text{Method}, F^? \mapsto \text{Field}]$

Path condition $\widehat{b} = \textbf{true}$ (not accumulated yet)

Figure 5.7: An example heap for an initial state of an execution

o, i.e. $m\left(\widehat{d}(o,c)\right) = \left\{o' \mid o \text{ owns}_h^+ o' \wedge \Gamma(o') \text{ gen}^* c\right\}$, where for two instances o, o' then o owns$_h$ o' iff there exists exactly one containment field $\widehat{f} \in \text{Field}_\blacklozenge$ such that $h(o,f) = o'$.

The symbolic type environment $\widehat{\Gamma}$ is consistent with the concrete type environment Γ if each symbolic expression (symbolic reference, symbolic reference set or symbolic instance) has a type bound that is consistent with the types assigned in Γ given mapping m; a type bound $(cs_{\text{in}}, cs_{\text{ex}})$ is consistent with a type c if there exist a $c' \in cs_{\text{in}}$ such that c is a subtype of c' (c gen$^*c'$) and there doesn't exist a $c'' \in cs_{\text{ex}}$ which c is a subtype of $(\neg\,(c \text{ gen}^* c''))$. Finally, logical constraints in $\widehat{b}$ are consistent with the model m if the expression $m(\widehat{b})$ we get by substituting all symbols in $\widehat{b}$ with m is true.

One symbolic heap *is stronger* than the other if all models (here all satisfying concrete heaps) of the former are also models of the latter. For conciseness, we let $(\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b_1}) \wedge \widehat{b_2}$ mean $(\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b_1} \wedge \widehat{b_2})$.

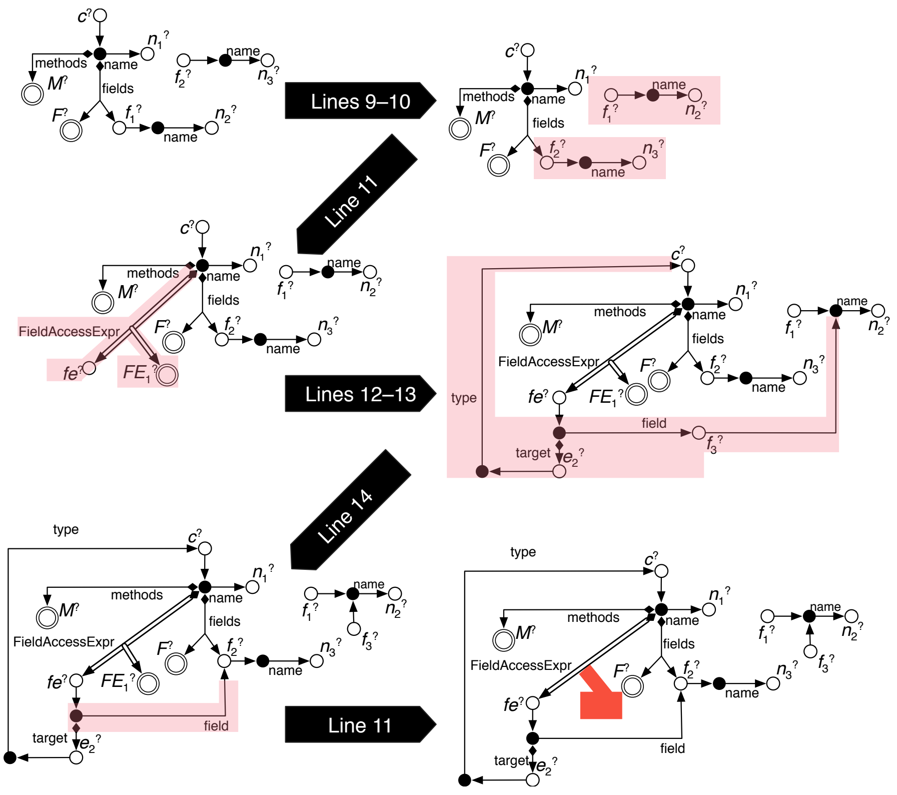


Figure 5.8: One of the paths when symbolically executing the example Rename-Field refactoring, starting with the symbolic state presented in Figure 5.7. Double-stroked arrows represent deep containment constraints.

Manipulating Symbolic State During Execution

Figure 5.8 shows an example path of the symbolic executor when executing the Rename-Field refactoring from Figure 5.2b starting with the symbolic state presented in Figure 5.7. The execution proceeds in the following steps:

- The initial statement on lines 9-10 replaces the old field (represented by symbol $f_1^?$) with the new field ($f_2^?$), such that the 'fields' reference of the target class ($c^?$) now points at $f_2^?$ instead of $f_1^?$. Remark, that changes are highlighted in red to ease understanding.
- Then we perform a deep matching on line 11, prompting the symbolic executor to create a deep containment constraint with type

FieldAccessExpr—represented by a double-stroked arrow ($\Rightarrow$)—for the location assigned to $c^?$; the containment constraint points to a symbolic set reference $FE_0^?$ (not shown in Figure) abstracting the set of all concrete instances of type FieldAccessExpr reachable from the concrete instance corresponding to $c^?$.

- In order to iterate over the elements of $FE_0^?$, we non-deterministically chose to partition it into disjoint symbol $fe^?$ and symbolic set $FE_1^?$, executing the body of the **foreach** with *faexpr* assigned to $fe^?$.
- To check the condition of the **if**-statement at lines 7-8, we perform a couple of field accesses, which triggers lazy initialization to creates two new symbolic instances: one which is assigned to the symbolic reference $fe^?$ and one which is assigned to its *target* field.
- We non-determinstically chose to execute the **then** branch—further constraining the values of the fields of $fe^?$—executing the field update statement at line 9, which updates the field access expression to point at the new renamed field instances.
- Finally, we are ready for another iteration at line 6, and this time non-deterministically chose to stop, further constraining $FE_1^?$ to be $\emptyset$ (thus disappearing in the figure).

We shall now define how these and other execution steps are realized. We start discussing the *basics* of the presentation format and the simple rules. Then we proceed to the four major ideas in our symbolic executor: *lazy initialization* during heap access and modification, *containment handling* when updating containment links, *lazy iteration* in **foreach**-loops, and *deep containment constraints* for handling matching expressions.

Basics. During the execution we maintain a store σ mapping variable names to symbolic set expressions. We use two symbolic evaluation functions for TRON's set ($\widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} = \widehat{e}$) and Boolean expressions ($\widehat{\mathcal{B}}[\![b]\!]\widehat{\sigma} = \widehat{b}$). They take concrete expressions with a store, and return resulting symbolic expressions by syntactically substituting all variables with their symbolic values as defined by σ. For example, we have $\widehat{\mathcal{B}}[\![x \subseteq y]\!][x \mapsto \{x^?\} \cup \{z^?\}, y \mapsto Y^?] = \{x^?\} \cup \{z^?\} \subseteq Y^?$.

The main judgement has the following format: $s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'$, denoting that the statement s evaluated in the symbolic store $\widehat{\sigma}$ and heap

$$\textsc{Skip}\ \frac{}{\mathbf{skip}, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}, \widehat{h}} \qquad \textsc{Seq}\ \frac{\begin{array}{c} s_1, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}'', \widehat{h}'' \\ s_2, \widehat{\sigma}'', \widehat{h}'' \longrightarrow \widehat{\sigma}', \widehat{h}' \end{array}}{s_1; s_2, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'}$$

$$\textsc{Agn}\ \frac{}{x := e, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}[x \mapsto \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma}], \widehat{h}}$$

$$\textsc{IfT}\ \frac{s_1, \widehat{\sigma}, \widehat{h} \wedge \widehat{\mathcal{B}}[\![b]\!]\widehat{\sigma} \longrightarrow \widehat{\sigma}', \widehat{h}'}{\mathbf{if}\, b\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'} \qquad \textsc{IfF}\ \frac{s_2, \widehat{\sigma}, \widehat{h} \wedge \neg\widehat{\mathcal{B}}[\![b]\!]\widehat{\sigma} \longrightarrow \widehat{\sigma}', \widehat{h}'}{\mathbf{if}\, b\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'}$$

$$\textsc{FixS}\ \frac{s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'' \qquad \widehat{h}' = \widehat{h}'' \wedge \left(\widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} = \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma}'\right)}{\mathbf{fix}\, e\, \mathbf{do}\, s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'}$$

$$\textsc{FixD}\ \frac{\begin{array}{c} s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}'', \widehat{h}''' \\ \widehat{h}'' = \widehat{h}''' \wedge \left(\widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} \neq \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma}''\right) \quad \mathbf{fix}\, e\, \mathbf{do}\, s, \widehat{\sigma}'', \widehat{h}'' \longrightarrow \widehat{\sigma}', \widehat{h}' \end{array}}{\mathbf{fix}\, e\, \mathbf{do}\, s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'}$$

$$\textsc{New}\ \frac{\begin{array}{c} x^?, \widehat{o}^\dagger \text{ fresh} \qquad \widehat{\sigma}' = \widehat{\sigma}[x \mapsto \{x^?\}] \qquad \widehat{z}' = \widehat{z}[x^? \mapsto \widehat{o}^\dagger] \\ \widehat{\Gamma}' = \widehat{\Gamma}[x^? \mapsto c, \widehat{o}^\dagger \mapsto c] \qquad \widehat{\ell}' = \widehat{\ell}[(\widehat{o}^\dagger, f) \mapsto \emptyset \mid f \in \text{fields}(c)] \end{array}}{x := \mathbf{new}\, c, \widehat{\sigma}, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b}) \longrightarrow \widehat{\sigma}', (\widehat{z}', \widehat{\ell}', \widehat{d}, \widehat{\Gamma}', \widehat{b})}$$

Figure 5.9: Symbolic execution rules for standard statements (σ is a symbolic variable store, h is a symbolic heap)

$\widehat{h}$, produces a new symbolic store $\widehat{\sigma}'$ and heap $\widehat{h}'$. The basic symbolic execution rules for statements are shown in Figure 5.9, including assignments, sequencing, branching, object creation and the **fix**-loop. These steps are essentially the same as in other existing symbolic executors. For straight-line statements the branch condition $\widehat{b}$ remains unchanged during execution (Skip, Agn, Seq, New). For branching statements it is amended (cf. IfT, IfF, FixS and FixD). Any execution continues as long as the heap is satisfiable, so satisfiability of $\widehat{h}$ is an implicit premise in all the rules. The branching rules are also non-deterministic; the non-determinism corresponds to branching (back-tracking) in the symbolic executor. Finally, the New rule, creates a new object of type c by allocating a symbolic instance and a symbolic reference $x^?$ pointing to it. All fields of the new instance are initialized to be empty sets.

Loops with unbounded iteration count give rise to infinite paths in symbolic execution (for instance by considering larger and larger inputs). In order for the symbolic execution algorithm to terminate, we bound the number of paths to be explored: FixD can only be applied a bounded number of times in a given execution of a given loop.

Lazy Initialization and Field Access. The rule for symbolic execution of field access is as follows:

$$\textsc{Acc}\;\dfrac{\begin{array}{c}\widehat{\text{singleton}}\left(\widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma},\widehat{h}\right) \ni (x^?,\widehat{h}'') \\ \widehat{\text{inst}}(x^?,\widehat{h}'') \ni (\widehat{o},\widehat{h}') \qquad \widehat{h}' = (\widehat{z}',\widehat{\ell}',\widehat{d}',\widehat{\Gamma}',\widehat{b}')\end{array}}{x := e.f,\widehat{\sigma},\widehat{h} \longrightarrow \widehat{\sigma}[x \mapsto \widehat{\ell}'(o,f)],\widehat{h}'}$$

Symbolically executing a field access $x := e.f$ requires the following steps:

1. Symbolically evaluating e to a symbolic set expression $\widehat{e}$.

2. Using $\widehat{\text{singleton}}$ function[3] to get a single symbol $x^?$ representing the value of $\widehat{e}$.

   a) If $\widehat{e}$ is not already a single symbol, then the $\widehat{\text{singleton}}$ function will generate a fresh symbol $x^?$ with the correct type and add the constraint $\widehat{e} = x^?$ to the heap, returning a new heap if satisfiable.

3. Lazily assigning a symbolic instance $\widehat{o}$ to $x^?$—if not already assigned—using the inst function, which non-deterministically either creates a new symbolic instance $\widehat{o}$ with the right type and shape, or picks an existing symbolic instance $\widehat{o}$ with compatible type bounds to treat aliasing.

4. Looking up the value of f of the assigned symbolic instance $\widehat{o}$ in the spatial part of the heap $\widehat{\ell}'$ assigning the resulting value to variable x.

[3]All our auxiliary functions are formally defined in Appendix B.

Containment and Field Updates. The symbolic execution of a field update statement $e_1.f := e_2$ follows a similar pattern to field access:

$$\textsc{Upd}\frac{\widehat{\text{singleton}}\left(\widehat{\mathcal{E}}[\![e_1]\!]\widehat{\sigma},\widehat{h}\right) \ni (x^?,\widehat{h}''') \qquad \text{inst}(x^?,\widehat{h}''') \ni (\widehat{o},\widehat{h}'') \qquad \widehat{\text{update}}(\widehat{o},f,\widehat{\mathcal{E}}[\![e_2]\!]\widehat{\sigma},\widehat{h}'') = \widehat{h}'}{e_1.f := e_2,\widehat{\sigma},\widehat{h} \longrightarrow \widehat{\sigma},\widehat{h}'}$$

After evaluating and resolving e to a symbolic instance $\widehat{o}$, the $\widehat{\text{update}}$ function is used to update field f of $\widehat{o}$ to point to the evaluated value of e_2 in the spatial constraints:

$$\widehat{\text{update}}(\widehat{o},f,\widehat{e},(\widehat{z},\widehat{\ell},\widehat{d},\widehat{\Gamma},\widehat{b})) = \begin{cases}(\widehat{z},\widehat{\ell}[(\widehat{o},f)\mapsto\widehat{e}],\widehat{d},\widehat{\Gamma},\widehat{b}) & \text{if } f \in \text{Field}_{\rightsquigarrow}\\ (\widehat{z},\widehat{\ell}'[(\widehat{o},f)\mapsto\widehat{e}],\widehat{d}',\widehat{\Gamma}',\widehat{b}\wedge\widehat{b}') & if\ f \in \text{Field}_{\blacklozenge}\end{cases}$$

$$\text{where } \widehat{\ell}' = \widehat{\text{disown}}(\widehat{e},\widehat{\ell})$$
$$(\widehat{d}',\widehat{\Gamma}',\widehat{b}') = \text{dc-}\widehat{\text{containment}}(\widehat{e},c,\widehat{z},\widehat{d},\widehat{\Gamma})$$

If f is a containment field we must further ensure that $\widehat{o}$ is the unique owner of $\widehat{e}$ which $\widehat{\text{update}}$ does by calling $\widehat{\text{disown}}$ and dc-$\widehat{\text{containment}}$.

$$\widehat{\text{disown}}(\widehat{e},\widehat{\ell}) = \left[(\widehat{o},f)\mapsto\widehat{\text{do-f}}(\widehat{e}',\widehat{e})\,\middle|\,(\widehat{o},f,\widehat{e}')\in\text{graph}\,\widehat{\ell}\right]$$

$$\text{where } \widehat{\text{do-f}}(\widehat{e}',\widehat{e}) = \begin{cases}\widehat{e}' & \text{if } f \in \text{Fields}_{\rightsquigarrow}\\ \widehat{e}'\setminus\widehat{e} & \text{if } f \in \text{Fields}_{\blacklozenge}\end{cases}$$

The $\widehat{\text{disown}}$ function presented above modifies each containment link in the spatial constraints $\widehat{\ell}$ to exclude the target symbolic expression $\widehat{e}$. The dc-$\widehat{\text{containment}}$ function analogously first excludes $\widehat{e}$ from all deep containment constraints to ensure that there are no stale references to the values of $\widehat{e}$, and then tries to correctly propagate the effects of the assignment of $\widehat{e}$ back to the containment constraints:

- For every containment constraint $\widehat{d}(\widehat{o},c) = \widehat{e}'$ with the same type as $\widehat{e}$ or a super-type of it, we generate a new set symbol $X^?$ with the target type, replace $\widehat{e}'$ with it and then add the constraint $X^? = \widehat{e}' \vee X^? = \widehat{e}\cup\widehat{e}'$ to the heap, which signifies that $\widehat{e}$ might have been added to the deep containment constraints of $\widehat{o}$; this highlights an interesting interaction between containment links and deep containment constraints which is not immediately obvious, but is necessary to maintain consistency while still keeping a high-level of symbolic abstraction.

- For subtypes of c, we do almost the same but use the constraint $(X^? = \widehat{e}' \vee X^? = \widehat{e}' \cup Y^?) \wedge \widehat{e} = Y^? \uplus Z^?$ instead, where $Y^?$ and $Z^?$ are fresh set symbols with $Y^?$ having type c and $Z^?$ having the type of $\widehat{e}$ excluding $\widehat{c}$ (in a type bound); this ensures that we refer to all elements in $\widehat{e}$ of type c and only those.

Lazy Iteration with First-class Set Expressions. We introduce *lazy iteration* over first-class symbolic set expressions. In particular, consider the operational rule for the **foreach**-loop below:

$$\textsc{For}\ \frac{\widehat{\text{init}}(me, \widehat{h}) = (e, \widehat{\varsigma}) \qquad x \hookleftarrow \widehat{\mathcal{E}}[\![e]\!]\widehat{\sigma} \vdash s, \widehat{\sigma}, \widehat{h}, \widehat{\varsigma} \xrightarrow{\text{each}} \widehat{\sigma}', \widehat{h}', \widehat{\varsigma}'}{\textbf{foreach}\ x \in me\ \textbf{do}\ s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'}$$

The first step is to use the $\widehat{\text{init}}$ function to get the expression e to be iterated over, and initialize a control state $\widehat{\varsigma}$ used during iteration depending on the kind of matching expression me provided; we will treat $\widehat{\varsigma}$ abstractly for now, and define it precisely later in this section. The other step is to use the $\xrightarrow{\text{each}}$-judgement to iterate over the values of $\widehat{e}$ depending on $\widehat{\varsigma}$, executing the **foreach**-body s at each iteration. The two rules for the $\xrightarrow{\text{each}}$-judgement, are provided below:

$$\textsc{ForB}\ \frac{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \widehat{\varsigma}) \ni (\text{break}, \widehat{h}')}{x \hookleftarrow \widehat{e} \vdash s, \widehat{\sigma}, \widehat{h}, \widehat{\varsigma} \xrightarrow{\text{each}} \widehat{\sigma}, \widehat{h}', \widehat{\varsigma}}$$

$$\textsc{ForC}\ \frac{\begin{array}{c}\widehat{\text{next}}(\widehat{e}, \widehat{h}, \widehat{\varsigma}) \ni (\text{cont}(x^?, \widehat{e}', \varsigma''), \widehat{h}''') \\ s, \widehat{\sigma}[x \mapsto x^?], \widehat{h}''' \longrightarrow \widehat{\sigma}'', \widehat{h}'' \qquad x \hookleftarrow \widehat{e}' \vdash s, \widehat{\sigma}'', \widehat{h}'', \widehat{\varsigma}'' \xrightarrow{\text{each}} \widehat{\sigma}', \widehat{h}', \widehat{\varsigma}'\end{array}}{x \hookleftarrow \widehat{e} \vdash s, \widehat{\sigma}, \widehat{h}, \widehat{\varsigma} \xrightarrow{\text{each}} \widehat{\sigma}', \widehat{h}', \widehat{\varsigma}'}$$

Both of the above rules depend on the $\widehat{\text{next}}$ function which given target expression $\widehat{e}$, current control state $\widehat{\varsigma}$ and heap $\widehat{h}$, provides a set of possible next actions; a possible action $\widehat{a}$ is either break which signals that iteration should stop, or is $\text{cont}(x^?, \widehat{e}', \widehat{\varsigma}')$ which signals that another iteration should happen with symbol $x^?$ bound to the range variable, $\widehat{e}'$ (disjoint from $x^?$) representing the rest of the values to be iterated over, and new control state $\widehat{\varsigma}'$. The first rule of the $\xrightarrow{\text{each}}$-judgement check whether break is a possible next state and if so it will stop iteration with new heap $\widehat{h}'$. The second rule checks whether cont is a possible next state, executes the loop body s with $x^?$ bound to the range variable x, and finally continues iteration over $\widehat{e}'$ in the updated states.

Now, observe how laziness is achieved with two key ideas: we never explicitly concretize *me*, leaving the level of concretization required to be decided by the $\widehat{\text{next}}$ function according to the control state $\widehat{\varsigma}$, and we iterate using a symbolic reference $x^?$ without requiring an assignment of a symbolic instance (to treat possible aliasing) as this point. Furthermore, by parameterizing the rules for **foreach** over functions $\widehat{\text{init}}$ and $\widehat{\text{next}}$, it would be easy to add new kinds of expressions without affecting the rules.

Type-Directed Matching with Containment Constraints. We will now discuss how the control state $\widehat{\varsigma}$ and functions $\widehat{\text{init}}$ and $\widehat{\text{next}}$ interact with matching expressions. We define the control state as follows:

$$\varsigma^? ::= \text{ns} \mid \text{ms}(c) \mid \text{ms}^*(c, \widehat{e}, \widehat{d})$$

Each alternative control state contains the required components to execute a given matching expression: ns is used for ordinary iteration, $\text{ms}(c)$ is used for shallow matching of elements against c and $\text{ms}^*(c, \widehat{e}, \widehat{d})$ is used for deep matching of elements against type c, a set of additional elements that must be iterated $\widehat{e}$ and a copy of the deep containment constraints $\widehat{d}$; the copy of containment constraints is kept in order to retrieve the deep containment constraint values that were available before iteration, which would represent the concrete objects that would have been matched by a concrete deep match operation at the time of evaluation. The $\widehat{\text{init}}$ function is therefore defined to map each expression to its initial control state:

$$\widehat{\text{init}}(e, \widehat{h}) = (e, \text{ns}) \quad \widehat{\text{init}}(e \ \textbf{match}\ c, \widehat{h}) = (e, \text{ms}(c))$$
$$\widehat{\text{init}}(e \ \textbf{match}^*\ c, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b})) = (e, \text{ms}^*(c, \emptyset, \widehat{d}))$$

The $\widehat{\text{next}}$ function is more interesting since it calculates the possible next actions for iteration. For sake of readability, we will define the $\widehat{\text{next}}$ function using inference rules, which describe the possible values in the result set. We define two rules for ordinary iteration:

$$\textsc{Iter-Emp}\frac{}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ns}) \ni (\text{break}, (\widehat{h} \wedge \widehat{e} = \emptyset))}$$

$$\textsc{Iter-Cont}\frac{\widehat{\text{partition}}(\widehat{e}, \widehat{h}) = (x^?, X^?, \widehat{h}')}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ns}) \ni (\text{cont}(x^?, X^?, \widehat{\text{ns}}), \widehat{h}')}$$

There are two possible actions: we can stop iterating if it is possible to constraint $\widehat{e}$ to be $\emptyset$ (Iter-Emp), and we can try to use $\widehat{\text{partition}}$ to split $\widehat{e}$ into a symbol $x^?$, and a disjoint set symbol $X^?$ which continue iteration with (Iter-Cont). The $\widehat{\text{partition}}$ function essentially generates fresh symbol $x^?$ and set symbol $X^?$ with the right types adding the constraint $\widehat{e} = x^? \uplus X^?$ to $\widehat{h}$.

For matching iterations, the rules for $\widehat{\text{next}}$ are defined as follows:

$$\textsc{IterM-Emp}\;\frac{}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ms}(c)) \ni (\text{break}, (\widehat{h} \wedge \widehat{e} = \emptyset))}$$

$$\textsc{IterM-Sucs}\;\frac{\widehat{\text{partition}}(\widehat{e}, \widehat{h}) = (x^?, X^?, \widehat{h''}) \qquad \widehat{\text{match}}(x^?, X^?, c, \widehat{h''}) \ni (\text{tt}, \widehat{h'})}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ms}(c)) \ni (\text{cont}(x^?, X^?, \text{ms}(c)), \widehat{h'})}$$

$$\textsc{IterM-Fail}\;\frac{\widehat{\text{partition}}(\widehat{e}, \widehat{h}) = (x^?, X^?, \widehat{h''}) \qquad \widehat{\text{match}}(x^?, X^?, c, \widehat{h''}) \ni (\text{ff}, \widehat{h'})}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ms}(c)) \ni (\text{break}, \widehat{h'})}$$

The terminating rule (IterM-Emp) is essentially unchanged, and the other two continuation rules use $\widehat{\text{partition}}$ to divide the target symbolic expression $\widehat{e}$ into a symbol $x^?$ and a symbolic reference set $X^?$. The symbol $x^?$ is then matched against $c^?$ using $\widehat{\text{match}}$ which returns a set of states each indicating whether matching $x^?$ against c was successful or not; if the match is successful (IterM-Sucs)—$(\text{tt}, \widehat{h'})$ is included in the result—then $\widehat{h'}$ constraints the type of $x^?$ to be a subtype of c, and otherwise if the match failed (IterM-Fail)—so $(\text{ff}, \widehat{h'})$ is included in the result—then $\widehat{h'}$ constraints the type bounds of $x^?$ and $X^?$ to exclude c as a possible supertype. A match is always successful if the type of $x^?$ is a subtype of c, always fails when the c is unrelated to or excluded from type bounds of $x^?$, and allows both when the type of $x^?$ is a supertype of c.

Finally, the rule for $\widehat{\text{next}}$ for deep matching delegates to an auxiliary definition $\widehat{\text{next-ms}}$ (IterMS):

$$\textsc{IterMS}\;\frac{\widehat{\text{next-ms}}(\widehat{e}, c, \widehat{e'}, \widehat{d}, \widehat{h}) \ni (\widehat{a}, \widehat{h'})}{\widehat{\text{next}}(\widehat{e}, \widehat{h}, \text{ms}^*(c, \widehat{e'}, \widehat{d})) \ni (\widehat{a}, \widehat{h'})}$$

There are three possible rules for $\widehat{\text{next-ms}}$, which are defined as follows:

$$\textsc{IterMSA-Emp}\;\frac{\widehat{\text{next}}(\widehat{e'}, \widehat{h} \wedge \widehat{e} = \emptyset, \text{ns}) \ni (\widehat{a}, \widehat{h'})}{\widehat{\text{next-ms}}(\widehat{e}, c, \widehat{e'}, \widehat{d}, \widehat{h}) \ni (\widehat{a}, \widehat{h'})}$$

$$\textsc{IterMSA-Fail}\,\frac{\begin{array}{cc}\widehat{\text{partition}}(\widehat{e},\widehat{h}) = (x^?, X^?, \widehat{h}'''') & \widehat{\text{match}}(x^?, X^?, c, \widehat{h}'''') \ni (\text{ff}, \widehat{h}''') \\ \widehat{\text{dcs}}(x^?, c, \widehat{d}, \widehat{h}''') \ni (\widehat{e}'', \widehat{h}'') & \widehat{\text{next-ms}}(X^?, \widehat{e}' \cup \widehat{e}'', \widehat{h}'') \ni (\widehat{a}, \widehat{h}')\end{array}}{\widehat{\text{next-ms}}(\widehat{e}, c, \widehat{e}', \widehat{d}, \widehat{h}) \ni (\widehat{a}, \widehat{h}')}$$

$$\textsc{IterMSA-Sucs}\,\frac{\begin{array}{c}\widehat{\text{partition}}(\widehat{e},\widehat{h}) = (x^?, X^?, \widehat{h}''') \\ \widehat{\text{match}}(x^?, X^?, c, \widehat{h}''') \ni (\text{tt}, \widehat{h}'') \quad \widehat{\text{dcs}}(x^?, c, \widehat{d}, \widehat{h}'') \ni (\widehat{e}'', \widehat{h}')\end{array}}{\widehat{\text{next-ms}}(\widehat{e}, c, \widehat{e}', \widehat{d}, \widehat{h}) \ni (\text{cont}(x^?, X^?, \text{ms}^*(c, \widehat{e}' \cup \widehat{e}'', \widehat{d})), \widehat{h}')}$$

The first rule constraints $\widehat{e}$ to $\emptyset$, and continues iterating over the symbolic set $\widehat{e}'$ which is used by the other rules to collect deep containment constraint values during iteration (IterMSA-Emp).

The second rule uses $\widehat{\text{partition}}$ and $\widehat{\text{match}}$ on c getting an unsuccessful match (IterMSA-Fail); we use the $\widehat{\text{dcs}}$ function to assign a location $\widehat{o}$ to $x^?$, lookup the deep containment constraint value $\widehat{d}(\widehat{o}, c) = \widehat{e}''$—creating it in the provided heap if non-existing)—and adding $\widehat{e}''$ to the control state. The rule then continues iterating over the rest $X^?$, since $x^?$ did not match the target type c and so must be skipped.

The third and final rule gets a successful match (IterMSA-Sucs) and so we use the $x^?$ for the next iteration; we still need to consider possible descendants of $x^?$ of type c and so use $\widehat{\text{dcs}}$ to get the deep containment constraint value and add it to the control state. Observe how the use of deep containment constraints allows us to provide a higher-level abstraction over structures focusing only on instances of the target type, and without explicitly considering all intermediate shapes of data.

Relating Concrete and Symbolic Semantics

To recover a deterministic semantics for programs from our provided non-deterministic operational symbolic semantics, one could define the symbolic semantics of a program as the set of all output pairs of symbolic stores and heaps obtained from non-deterministically executing each different feasible path in the program.

$$\widehat{\mathcal{S}}[\![s]\!](\widehat{\sigma}, \widehat{h}) = \left\{ (\widehat{\sigma}', \widehat{h}') \middle| s, \widehat{\sigma}, \widehat{h} \longrightarrow \widehat{\sigma}', \widehat{h}'' \right\}$$

A useful property to show is then that the deterministic symbolic semantics is sound with regards to the concrete semantics, i.e.:

Theorem 5.1 (Soundness). *If* $\exists m.\sigma = m(\widehat{\sigma}) \wedge \Gamma, h \overset{m}{\models} \widehat{h}$ *and* $\widehat{\mathcal{S}}[\![s]\!](\widehat{\sigma}, \widehat{h}) = \widehat{\mathbf{M}}$ *then for all* $(\widehat{\sigma}, \widehat{h}) \in \widehat{\mathbf{M}}$ *there exists* σ', Γ', *and* h' *such that we have a concrete execution* $s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'$ *and exists a model* m' *such that* $\sigma' = m'(\widehat{\sigma}')$ *and* $\Gamma', h' \overset{m'}{\models} \widehat{h}'$.

5.4 Evaluation

We implemented our technique in a prototype tool,[4] which we evaluate to show the concrete benefits of our technique.

Test Generation

White-box test generation is a classical application of symbolic execution, and we aim to use it as an example for evaluating our symbolic execution algorithm. We have built a white-box test generator and compare its effectiveness against a baseline black-box test generator.

We aim to compare the test generators according to their effectiveness, which is how well a test suite exercises the transformation-under-test (TUT). We use branch coverage as the target metric, which we define for TRON constructs as follows: for `if`-statements both branches must be taken, for `fix`-loops we check whether it is run one or more times[5] and for `foreach`-loops we check whether it is run zero, one or more times.

White-box Test Generator. The white-box test generator is a simple extension of the symbolic execution algorithm presented in Section 5.3, requiring only two new additions:

1. Memoising a copy of the spatial constraints whose values are not modified by field updates, thus keeping track of the initial structure of input.
2. A translator between the output model given by the model finder to concrete data usable by the target TRON program.

[4]`https://github.com/itu-square/SymexTRON`
[5]`fix`-loops must run the body at least once, according to the semantics

Black-box Test Generator. The black-box test generator optimizes towards *meta-model coverage*, which is the literature-recommended metric [Wang et al., 2006; Finot et al., 2013]. A test suite is said to have full meta-model coverage if each subtype of relevant classes is present in at least in one test case, and each relevant field is instantiated with each valid multiplicity (i.e. zero, one or many).

Subject Programs. The subject programs were selected according to three criteria: be an interesting representative variety of realistic transformations, be independently specified to avoid bias, and be feasible to implement in TRON. To fulfill the first criterion, we chose transformations from two categories: model transformations and refactorings. For the second criterion, we ported the model transformations from the ATL transformation zoo[6] and chose the refactorings from Fowler's classic collection [Fowler, 1999]. The third criterion is achieved by picking suitably sized transformations that satisfy our resource and design constraints, since it takes time to manually port complex transformations correctly (despite language similarity) and TRON lacks abstractions for modularity that full languages have. We ended up with 3 model transformations and 4 refactorings, all of which we describe below.

Refactorings.

- An extended version of the *Rename field* refactoring used as the running example (see Figure 5.2).
- *Rename method*: renames a target method in a class, and ensures that all calls to the correct overloading of this method (with the right types) must refer to the updated name.
- *Extract superclass*: creates a common superclass of two classes with similar structure ensuring that all common fields are pulled up and that both classes inherit from it.
- *Replace delegation with inheritance*: Makes a class inherit directly from a type instead of using a field for delegation, updating all method calls targeting that field to use **this** as a target instead.

[6] `https://www.eclipse.org/atl/atlTransformations/`

Model Transformations.

- The *Families to Persons* model transformation (Fam2Pers), converts a model of a traditional family with a mother, father and possibly children to a collection of individuals with explicit gender (male or female).
- The classical *Class to Relational* model transformation (Class2Rel), which converts an object-oriented class model to a relational database schema.
- The *Path expression to Petri net* model transformation (Path2Petri), which converts a path expression with states and transitions to a full Petri net with named places, different types of arcs and weighted transitions.

The characteristics and basline meta-model coverage is presented in Table 5.2, and the source code of all the programs is available in Appendix D.

Porting Transformations to TRON. By design TRON is a minimal language, and so there are non-core transformation languages features that must be handled when porting transformations from fully-featured languages to TRON. In particular, three features had to be handled for the considered subject programs: functions, implicit tracing links and circular data dependencies (the latter two present in model transformation languages like ATL). When a transformation is ported to TRON one must take care to correctly inline function calls, which is done by replacing the calls with the function body, substituting the parameters with the provided arguments, renaming local variables to avoid clashes, and converting any explicit recursion to use the '**foreach**'-statement or '**fix**'-statement. To handle tracing links one must take care to augment the meta-model to include them explicitly, and to explicate assignment to the tracing links on object creation in the transformation; for circular data dependencies one must ensure to initialize all relevant objects before the transformation.

Set-up. The experiment was set-up to automatically run both test generators automatically on all the described subject programs. The white-box test generator was bounded in the number of iterations (2, except

Program	LOC	Meta-model coverage (%)	
		Black-box	White-box
RenameField	53 + 12 = 65	96.55	70.69
RenameMethod	53 + 28 = 81	96.55	93.10
ExtractSuper	53 + 29 = 82	98.28	31.03
ReplaceDelegation	53 + 30 = 83	100.00	85.19
Fam2Pers	21 + 56 = 77	100.00	88.00
Path2Petri	42 + 58 = 100	100.00	37.50
Class2Rel	34 + 100 = 134	100.00	100.00

Table 5.2: Characteristics and basline meta-model coverage for the subject programs. Here LOC indicates lines of code, where the first component of the summation is the size of the data model and the second component is the size of the transformation.

for Fam2Pers which uses 3 due to meta-model constraints) and instances considered by the model finder (6 for model transformations, 10 for refactorings), and a time-out of 1 hour was put in place. We also added light-weight support for bidirectional fields in the model finder to better support the model transformations. The prototype symbolic executor, was implemented in Scala 2.11.7 [Odersky and Rompf, 2014] and the evaluation was run on a 2.3 GHz Core i7 MacBook Pro (OS X 10.11). The external model finder KodKod was configured to use the parallel SAT solver Plingeling [Biere, 2014].

Test Generation Results. We ran a series of toy programs exercising the various constructs of TRON as a warm up for our test generators; the white-box test generator achieved 100% code coverage for all programs beating the black-box test generator, and all under 30 seconds of execution time.

Table 5.3 shows the results of running the test generators on the selected subject programs (model transformations and refactorings).

Refactorings. The white-box test generator achieves better code coverage than the baseline black-box test generator for all the refactorings, reaching 100% coverage for two. We hypothesise that refactorings do many targeted modifications of complex models, making it hard to

Program	Branch coverage (%)		Time (s)	
	Black-box	White-box	Black-box	White-box
RenameField	60.00	100.00	141.5	336.6
RenameMethod	26.67	93.33	141.7	3600.0
ExtractSuper	75.00	100.00	124.8	386.4
ReplaceDelegation	47.06	76.47	115.5	3600.0
Fam2Pers	100.00	100.00	4.8	135.4
Path2Petri	88.89	33.33	1.8	3600.0
Class2Rel	70.83	75.00	3.8	3600.0

Table 5.3: Results of running the test generators on subject programs. The branch coverage indicates the percentage of the total reachable branches covered, and the time indicates the execution time of the tool from start to finish on the evaluation subjects.

generate tests that cover the required paths without access to the transformation code; The test generated by black-box had high meta-model coverage (see Table 5.2) as expected, but achieved low branch coverage since it did not fully exercise the transformation (Table 5.3); this is in contrast to the white-box test generator which generated more focused test cases.

Due to the nature of symbolic execution, the white-box test generator was slower than the black-box test generator. We believe that the higher precision in bug finding offsets the runtime cost; test generation is an occasional offline task and 1 computer hour is not unreasonable to use compared to the many hours a programmer would otherwise have spent on the same task.

Model Transformations. The white-box test generator achieved good results for model transformations, performing better than the black-box test generator for the Fam2Pers and Class2Rel transformations, and worse for the Path2Petri transformation. We suspect that the black-box test generator performs well on model transformations because they primarily translate structured data according to their meta-model types, without relying on complex constraints and cases. Therefore, having a test suite with high meta-model coverage will generate the necessary different types to trigger the right execution paths resulting in acceptable branch coverage. The white-box test generator performed less impressively on model transformations than refactorings because the symbolic executor did not reach the right paths before the time-out triggered. The sequential composition of complex for-loops results in an explosion of paths to be explored, and so it takes significantly more time to explore the whole program and generate interesting tests.

Comparing Coverage Criteria. The black-box test generator optimizes towards achieving maximal meta-model coverage (see Table 5.3), but seems to achieve mixed branch coverage results, and although it performed better for model transformations than for refactorings, it never reached full branch coverage. This indicates that there is little guarantee that high meta-model coverage ensures high branch coverage, which is an interesting experience for building both white-box and black-box tools.

The symbolic executor achieved high branch coverage for most subjects without achieving the same high meta-model coverage—the lowest being 31.03% meta-model coverage for the ExtractSuper refactoring that we achieved full branch coverage for—which indicates that there is no correlation the other way as well. It would of course require a more extensive empirical study to conclusively affirm our hypothesis.

Comparison with Symbolic Executors for Object-Oriented Languages

Two of the best known and well-supported symbolic executors for object-oriented programming languages are Symbolic PathFinder [Pasareanu et al., 2013] and Microsoft IntelliTest/Pex [Tillmann and de Halleux, 2008]. We will describe our experiences trying to encode various high-level transformation features—sets, containment and deep matching—and the difficulties faced when these features are not handled first-class.

Symbolic Support for Sets. The first challenge we faced was how to symbolically encode sets in traditional symbolic executors. Symbolic PathFinder does not directly support standard Java collections like HashSet or TreeSet, and using those collections during symbolic execution leads to errors: a symbol does not have a hash value, and it is not possible to symbolically compare two objects. One could try a more cumbersome encoding by using lists and handling inequality constraints explicitly, but it is unclear how to generate interesting instances of such sets automatically. Pex tries to dynamically construct instances of sets using arrays, but it is hard in practice to make it generate an array of distinct elements that is usable to construct an interesting set.

We treat set values first-class which exploits the support of set theories in model finders like KodKod and SMT solvers [Kröning et al., 2009]. We believe that implementing this could be beneficial for these traditional symbolic executors as well.

Enforcing Deep Containment Constraints. Another challenge is enforcing containment-like constraints with acyclicity and non-sharing for abstract syntax trees. Traditional symbolic executors support deep containment constraints neither directly nor indirectly. Hypothetically, acyclicity constraints could be enforced statically by giving all objects unique identifiers and fixing an ordering between contained objects and parents; however, this requires both the management of a complex system on top of existing dynamic structures, and it is unclear how to efficiently handle dynamic to such links. In contrast, first class support of these constraints in TRON makes it easy to handle dynamic updates and allows specialized techniques to be used to handle such constraints.

Deep Matching and Visitors. Deep matching is straightforward to encode using reflection, but reflection is not handled well by symbolic executors, and so is to be avoided.

Traditionally traversal of abstract syntax is done using recursive visitors, which uses plain classes and thus is better supported. However, this approach is non-optimal from a symbolic execution point of view, since to reach the relevant part of an abstract syntax tree—like a field access expression—one has to consider and enumerate all intermediate shapes—i.e., classes, methods, different kinds of statements and containing expressions in our example—which hits a combinatorial explosion, even with reasonably small bounds.

In contrast, we abstract away intermediate shapes with deep containment constraints, which allows reasoning about only the parts of the data structure we are interested in. Transformations like refactorings often perform local changes on the abstract syntax trees, and so this approach seems especially beneficial in those cases.

Threats and Limitations

The main threat to validity of the experiment is that we implemented the subject programs ourselves, introducing the possibility of bias and errors in the implementation. For the model transformations, we mitigated this by choosing existing ones from ATL, and for the refactorings we chose a number of standard ones from Fowler's authoritative book [Fowler, 1999]. Furthemore, minor implementation mistakes are of lesser importance since the number of found errors is not an evaluation criterion.

Inozemtseva and Holmes [2014] show that test coverage is not a strongly correlative measure for effectiveness. However, arguably a test suite which has low code coverage is going to miss bugs because it simply does not visit code present in some of the branches. The black-box test generator has been implemented by us optimizing for the standard meta-model coverage metric [Finot et al., 2013; Wang et al., 2006] to avoid bias, since we could not find an existing third-party tool that was suitable for our purposes.

We are not experts on Symbolic PathFinder and Pex, and could have missed better ways to encode high-level features. We mitigated this by systematically reading the available documentation, and searching on forums and mailing lists for answers to similar challenges.

5.5 Related Tools and Approaches

Symbolic Execution of High-level Transformation Languages. Simple symbolic execution algorithms [Lucio and Vangheluwe, 2013] exist for significantly less expressive transformation languages like DSLTrans, which bounds loops and does not permit dependent state and loop iterations. This lack of expressiveness allows the symbolic executor to be heavily specialized and quick, but the algorithm is hard to generalize for more expressive Turing-complete languages like TRON.

More complex whitebox-based algorithms are presented by ATL-Test [González and Cabot, 2012] and TETRA Box [Schönböck et al., 2013]. These tools only support a class of transformations that can not modify input state. Therefore, it is not possible to easily express complex transformations like the refactorings considered in this paper. Furthermore, the method presented in this paper, is fully-formalized and evaluated, showing applicability of our framework for a broader range of transformations.

Test Generation for Transformations. The latest survey on verification of model transformations [Rahim and Whittle, 2015] shows that most test generation techniques for model transformations focus on black-box testing, which do not account for concrete transformation semantics and thus may fail to cover program statements as shown in our evaluation. There is a test generation tool for Maude [Riesco, 2010, 2012] based on bounded narrowing (practically, symbolic execution for rewriting languages). However, complex transformations are hard to write in the style of term-rewriting systems—since they modify object graphs—and the object-oriented extension is not as far as we understand supported by the test generator. A black-box test generation tool called Dolly [Mongiovi et al., 2014] is used to test C and Java based refactoring engines with promising results. As our evaluation results indicate that white-box based techniques have better effectiveness than black-box based ones, it could be interesting to see whether we could adapt some of our novel ideas for a language like Java and increase the number of bugs found.

Dynamic Symbolic Execution Our evaluation shows that it is relatively cheap to generate random initial test input with acceptable coverage. We believe that it might be fruitful to look at hybrid white-box approaches like Dynamic Symbolic Execution [Korel, 1990; Godefroid et al., 2005] to increase performance and scalability for larger transformations.

5.6 Recap

This chapter presented a formal symbolic execution algorithm for a small formal transformation language TRON capturing key high-level transformation features from Chapter 2. In order to feasibly model the features symbolically without hitting a combinatorial explosion, we had to combine ideas from existing work, i.e., lazy initialization and symbolic reference sets, with novel ideas that were targeted towards the modelled high-level features, i.e. containment handling, lazy iteration and deep containment constraints. The symbolic technique was implemented in a prototype white-box test generation tool, which compared favorably against the baseline black-box test generator, achieving high test coverage for virtually all the selected subject transformation. This shows great promise in adapting programming language techniques to transformation languages, and made it possible for us in the later chapters to tackle a large subset of Rascal by using the techniques and experiences we gathered.

Chapter 6

The Formal Semantics of Rascal Light

Rascal [Klint et al., 2009, 2011] is a high-level transformation language that aims to simplify software language engineering tasks like defining program syntax, analyzing and transforming programs, and performing code generation. The language provides several features including built-in collections (lists, sets, maps), algebraic data-types, powerful pattern matching operations with backtracking, and high-level traversals supporting multiple strategies.

Interaction between different language features can be difficult to comprehend, since most features are semantically rich. My goal is to provide a well-defined formal semantics for a large subset of Rascal called Rascal Light—in the spirit of (core) Caml Light [Leroy, 1997], Clight [Blazy and Leroy, 2009], and Middleweight Java [Bierman et al., 2003]—suitable for developing formal techniques, e.g., type systems and static analyses.

Scope

Rascal Light aims to model a realistic set of high-level transformation language features, by capturing a large subset of the Rascal operational language. Unlike the small formal transformation language TRON (Chapter 5), Rascal Light targets being practically usable, i.e., it should be possible to translate many realistic pure Rascal programs to this subset by an expert programmer without losing the high level of abstraction. The following Rascal features are captured in Rascal Light:

- Fundamental definitions including algebraic data-types, functions with type parameters, and global variables.

- Basic expressions, including variable assignment, definition and access, exceptions, collections (including sets, lists and maps), **if**-expressions, **switch**-statements, **for**-loops and **while**-loops including control flow operators.
- Powerful pattern matching operations including type patterns, non-linear pattern matching, (sub)collection patterns, and descendant pattern matching.[1]
- Backtracking using the **fail** operator, including roll-back of state.
- Traversals using generic **visit**-expressions, supporting the different kinds of available strategies: **bottom-up**(**-break**), **top-down**(**-break**), **innermost**, and **outermost**.
- Fixed-point iteration using the **solve**-loop.

The following Rascal features are considered out of scope:

- Concrete syntax declaration and literals, string interpolation, regular expression matching, date literals and path literals
- The standard library including Input/Output and the Java foreign function interface
- The module system, include modular extension of language elements such as datatypes and functions
- Advanced type system features, like parametric polymorphism, the numerical hierarchy and type inference

In Rascal, Boolean expressions can contain pattern matching subexpressions and backtracking is affected by the various Boolean operators (conjunction, disjunction, implication). In Rascal Light, backtracking is more restricted, which I believe heavily simplifies the semantics while losing little expressiveness in practice; most programs could be rewritten to support the required backtracking at the cost of increased verbosity.

Method

I have used the language documentation,[2] and the open source implementation of Rascal,[3] to derive the formalism. The syntax is primarily

[1]Similar to type-directed querying in TRON
[2]`http://tutor.rascal-mpl.org/Rascal/Rascal.html`
[3]`https://github.com/usethesource/rascal`

based on the μRascal syntax description [CWI Amsterdam, 2017], but altered to focus on the high-level features. The semantics is based on the description of individual language features in the documentation. In case of under-specification or ambiguity, I used small test programs to check what the expected behavior is. I thank the Rascal developers for our personal correspondence which further clarified ambiguity and limitations of the semantics compared to Rascal.

To discover possible issues, the semantics has been implemented as a prototype and tested against a series of Rascal programs. To strengthen the correctness claims, I have proven a series of theorems of interest in Section 6.3, since as Milner et al. [1990] states:

The robustness of the semantics depends upon theorems

— The Definition of Standard ML

6.1 Abstract Syntax

Rascal Light programs are organized into modules that consist of definitions of global variables, functions and algebraic data types. I will in the rest of this chapter assume that modules are well-formed: top-level definitions and function parameters have unique names, all function calls, constructors and datatypes used have corresponding definitions, and all variables are well-scoped with no shadowing. To maintain a clean presentation, I will not write the definition environment explicitly, but mention the necessary definitions as premises when required.

Rascal Light has three kinds of definitions: global variables, functions and algebraic data-types. Global variables are typed and are initialized with a target expression at module loading time. Functions have a return type, a unique name, a list of typed uniquely named parameters, and have an expression as a body. Algebraic data-types have unique names and declare a set of possible alternative constructors, each taking a list of typed fields as arguments.

$$\begin{aligned}
d ::= {} & \mathbf{global}\ t\ y = e && (\textit{Global variables}) \\
\mid {} & \mathbf{fun}\ t\ f(\underline{t\ x}) = e && (\textit{Function Definition}) \\
\mid {} & \mathbf{data}\ at = k_1(\underline{t\ x}) \mid \ldots \mid k_n(\underline{t\ x}) && (\textit{Algebraic Data Type Definition})
\end{aligned}$$

The operational part of Rascal consists of syntactically distinct categories of statements and expressions; I have chosen to collapse the two categories in Rascal Light since statements also have result values that can be used inside expressions. Most constructs are standard imperative ones, such as blocks, assignment, branching and loops (including control operations); I have chosen to use **local-in** for representing blocks in Rascal Light instead of curly braces like in Rascal, to distinguish them from set literal expressions; blocks contain locally declared variables and a sequence of expressions to evaluate.

Notable Rascal-specific constructs include a generalized version of the **switch**-expression that supports rich pattern matching over both basic values and algebraic data types, the **visit**-expression which allows traversing data types using various strategies (Example 6.1), and the **solve**-expression which continuously evaluates its body expression until target variables reach a fixed-point in assigned values (Example 6.2). The **fail** control operator allows backtracking inside **switch** and **visit** statements to try other possible matches (Example 6.3).

Example 6.1 (Expression simplifier). An expression simplifier can use visit to completely simplify all sub-expressions no matter where they are in the input expression:

```
data Expr = intlit(int v) | plus(Expr lop, Expr rop) | ...;

Expr simplify(Expr e) =
  bottom-up visit(e) {
    case plus(intlit(0), y) => y
    case plus(x, intlit(0)) => x
  };
```

▲

Example 6.2 (Lattice fixed-point). The Kleene fixed-point of a continuous function in a lattice for a domain *Val* with functions for the bottom element and least-upper bound can be computed using solve:

```
Val fix(Fun f) = {
   Val v = bottom();
   solve(v) {
      v = lub(v, apply(f, v))
   };
}
```

▲

Example 6.3 (Knapsack problem). The knapsack problem concerns finding a subset of items with greatest value under a specified maximum weight. The function below uses backtracking to (slowly) find the optimal solution.

```
set[Item] slowknapsack(set[Item] items, int maxWeight) = {
  set[Item] res = {};
  solve(res) {
    switch(items) {
      case {*xs, *ys}:
      if (sumweights(xs) > maxWeight
          || sumvalues(xs) < sumvalues(res)) fail;
      else res = xs;
    };
    res
  };
}
```

▲

$$
\begin{array}{rll}
e ::= & vb & (\textit{Basic Values }, vb \in \{1, \text{“}\texttt{foo}\text{”}, 3.14, \ldots\}) \\
| & x & (\textit{Variables}, x, y \in \mathrm{Var}) \\
| & \ominus\, e & (\textit{Unary Operations}, \ominus \in \{-, \ldots\}) \\
| & e_1 \oplus e_2 & (\textit{Binary Operations}, \oplus \in \{+, -, \times, \ldots\}) \\
| & k(\underline{e}) & (\textit{Constructor Applications}, k \in \mathrm{Constructor}) \\
| & [\underline{e}] & (\textit{List Expressions}) \\
| & \{\underline{e}\} & (\textit{Set Expressions}) \\
| & (\underline{e : e'}) & (\textit{Map Expressions}) \\
| & e_1[e_2] & (\textit{Map Lookup}) \\
| & e_1[e_2 = e_3] & (\textit{Map Update}) \\
| & f(\underline{e}) & (\textit{Function Calls}, f \in \mathrm{Function}) \\
| & \mathbf{return}\ e & (\textit{Return Expressions}) \\
| & x = e & (\textit{Variable Update Assignments}) \\
| & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & (\textit{Conditionals}) \\
| & \mathbf{switch}\ e\ \mathbf{do}\ \underline{cs} & (\textit{Case Matching}) \\
| & st\ \mathbf{visit}\ e\ \mathbf{do}\ \underline{cs} & (\textit{Deep Traversal}) \\
| & \mathbf{break} \mid \mathbf{continue} \mid \mathbf{fail} & (\textit{Control Operations}) \\
| & \mathbf{local}\ \underline{t\ x}\ \mathbf{in}\ \underline{e}\ \mathbf{end} & (\textit{Blocks}) \\
| & \mathbf{for}\ g\ e & (\textit{Iteration}) \\
| & \mathbf{while}\ e\ e & (\textit{While Expressions}) \\
| & \mathbf{solve}\ \underline{x}\ e & (\textit{Fixedpoint Expressions}) \\
| & \mathbf{throw}\ e & (\textit{Exception Throwing})
\end{array}
$$

$$
\begin{array}{rll}
& \mid \textbf{try}\ e_1\ \textbf{catch}\ x \Rightarrow e_2 & (\textit{Exception Catching}) \\
& \mid \textbf{try}\ e_1\ \textbf{finally}\ e_2 & (\textit{Exception Finalization}) \\
cs ::= & \textbf{case}\ p \Rightarrow e & (\textit{Case})
\end{array}
$$

There are two kinds of *generator expressions*: *enumerating assignment* where the range variable is assigned to each element in the result of a collection-producing expression, and *matching assignment* that produces all possible assignments that match target patterns (defined later).

$$
\begin{array}{rll}
g ::= & x \leftarrow e & (\textit{Enumerating Assignment}) \\
& \mid p := e & (\textit{Matching Assignment})
\end{array}
$$

The **visit**-expression supports various strategies that determine the order a particular value is traversed w.r.t. its contained values. The **top-down** strategy traverses the value itself before contained values, and conversely **bottom-up** traverses contained values before itself. The **break** versions stop the traversal at first successful match, and the **outermost** and **innermost** respectively apply the **top-down** and **bottom-up** until a fixed-point is reached.

$$
\begin{array}{rll}
st ::= & \textbf{top-down} & (\textit{Preorder Traversal}) \\
& \mid \textbf{bottom-up} & (\textit{Postorder Traversal}) \\
& \mid \textbf{top-down-break} & (\textit{First-match Preorder Traversal}) \\
& \mid \textbf{bottom-up-break} & (\textit{First-match Postorder Traversal}) \\
& \mid \textbf{outermost} & (\textit{Fixedpoint Preorder Traversal}) \\
& \mid \textbf{innermost} & (\textit{Fixedpoint Postorder Traversal})
\end{array}
$$

Like Rascal, Rascal Light has a rich pattern language that not only includes matching on basic values and constructors, but also powerful matching inside collections and descendant patterns that allow matching arbitrarily deeply contained values.

$$\begin{aligned}
p ::= {} & vb && (\textit{Basic Value Patterns}) \\
\mid {} & x && (\textit{Variable Patterns}) \\
\mid {} & k(\underline{p}) && (\textit{Deconstructor Patterns}) \\
\mid {} & t\ x : p && (\textit{Typed Labelled Patterns}) \\
\mid {} & [\underline{\star p}] && (\textit{List Patterns}) \\
\mid {} & \{\underline{\star p}\} && (\textit{Set Patterns}) \\
\mid {} & !p && (\textit{Negation pattern}) \\
\mid {} & /p && (\textit{Descendant Patterns})
\end{aligned}$$

Patterns inside collections can be either ordinary patterns or *star patterns* that match a subcollection with arbitrary number of elements.

$$\begin{aligned}
\star p ::= {} & p && (\textit{Ordinary Pattern}) \\
\mid {} & \star x && (\textit{Star Pattern})
\end{aligned}$$

Rascal Light programs expressions evaluate to values, which either are basic values, constructor values, collections or the undefined value (■).

$$\begin{aligned}
v ::= {} & vb && (\textit{Basic Values}) \\
\mid {} & k(\underline{v}) && (\textit{Constructor Values}, k \in \text{Constructor}) \\
\mid {} & [\underline{v}] && (\textit{List Values}) \\
\mid {} & \{\underline{v}\} && (\textit{Set Values}) \\
\mid {} & (\underline{v : v'}) && (\textit{Map Values}) \\
\mid {} & \blacksquare && (\textit{Undefined Value})
\end{aligned}$$

6.2 Semantics

I present a formal development of the dynamic aspects of Rascal Light, using a natural semantics specification. Natural (big-step) semantics [Kahn, 1987] is particularly suitable for Rascal, because it closely mimics semantics of an interpreter and the high-level features—exceptions, backtracking, traversals—introduce a rich control-flow that depends not only on the structure of the program but also on the provided input. There is no concurrency or interleaving in Rascal, and so there is no need for a more fine-grained operational semantics like (small-step) structural operational semantics (SOS) [Plotkin, 2004].

Value Typing

Rascal (Light) is strongly typed and so all values are typed. The types are fairly straight-forward in the sense that most values have a canonical type and there is a sub-typing hierarchy (explained shortly) with a bottom type **void** and a top type **value**.

$$
\begin{aligned}
t ::= {} & tb & (\textit{Base Types}, tb \in \{\text{Int}, \text{Rational}, \text{String}, \ldots\}) \\
\mid {} & at & (\textit{Algebraic Data Types}, at \in \text{DataType}) \\
\mid {} & \mathbf{set}\langle t \rangle & (\textit{Sets}) \\
\mid {} & \mathbf{list}\langle t \rangle & (\textit{Lists}) \\
\mid {} & \mathbf{map}\langle t_1, t_2 \rangle & (\textit{Maps}) \\
\mid {} & \mathbf{void} & (\textit{Bottom Type}) \\
\mid {} & \mathbf{value} & (\textit{Top Type})
\end{aligned}
$$

I provide a typing judgment for values of form $\boxed{v : t}$, which states that value v has type t. Basic values are typed by their defining basic type, the undefined value (■) has the bottom type **void**, and constructor values are typed by their corresponding data type definition assuming the contained values are well-typed. The type of collections is determined by the contained values, and so a least upper bound operator is defined in types—following the sub-type ordering—which is used to infer a precise type for the value parameters.

$$
\textsc{T-Basic}\,\frac{vb \in [\![tb]\!]}{vb : tb} \qquad \textsc{T-Bot}\,\frac{}{\blacksquare : \mathbf{void}}
$$

$$
\textsc{T-Constructor}\,\frac{\mathbf{data}\ at = \ldots \mid k(\underline{t}) \mid \ldots \qquad \underline{v : t'} \qquad \underline{t' <: t}}{k(v_1, \ldots, v_n) : at}
$$

$$
\textsc{T-Set}\,\frac{\underline{v : t}}{\{\underline{v}\} : \mathbf{set}\langle \bigsqcup \underline{t} \rangle} \qquad \textsc{T-List}\,\frac{\underline{v : t}}{[\underline{v}] : \mathbf{list}\langle \bigsqcup \underline{t} \rangle} \qquad \textsc{T-Map}\,\frac{\underline{v : t} \qquad \underline{v' : t'}}{(\underline{v : v'}) : \mathbf{map}\langle \bigsqcup \underline{t}, \bigsqcup \underline{t'} \rangle}
$$

The subtyping relation has form $\boxed{t <: t'}$, stating t is a subtype of t'. We let $\boxed{t \not<: t'}$ denote the negated form where none of the given cases below matches. Sub-typing is reflexive, so every type is a sub-type of itself; **void** and **value** act as bottom type and top type respectively.

$$
\textsc{ST-Refl}\,\frac{}{t <: t} \qquad \textsc{ST-Void}\,\frac{}{\mathbf{void} <: t} \qquad \textsc{ST-Value}\,\frac{}{t <: \mathbf{value}}
$$

Collections are covariant in their type parameters, which is safe since all values are immutable.

$$\text{ST-LIST}\ \frac{t <: t'}{\mathbf{list}\langle t\rangle <: \mathbf{list}\langle t'\rangle} \qquad \text{ST-SET}\ \frac{t <: t'}{\mathbf{set}\langle t\rangle <: \mathbf{set}\langle t'\rangle}$$

$$\text{ST-MAP}\ \frac{t_{\text{key}} <: t'_{\text{key}} \qquad t_{\text{val}} <: t'_{\text{val}}}{\text{map}\langle t_{\text{key}}, t_{\text{val}}\rangle <: \mathbf{map}\langle t'_{\text{key}}, t'_{\text{val}}\rangle}$$

The least upper bound on types is defined as follows:

$$\bigsqcup \varepsilon = \mathbf{void} \qquad \bigsqcup t, \underline{t'} = t \sqcup \bigsqcup \underline{t'}$$

$$t_1 \sqcup t_2 = \begin{cases} t_1 & \mathbf{if}\ t_2 = \mathbf{void} \lor t_1 = t_2 \\ t_2 & \mathbf{if}\ t_1 = \mathbf{void} \\ \mathbf{list}\langle t'_1 \sqcup t'_2\rangle & \mathbf{if}\ t_1 = \mathbf{list}\langle t'_1\rangle \land t_2 = \mathbf{list}\langle t'_2\rangle \\ \mathbf{set}\langle t'_1 \sqcup t'_2\rangle & \mathbf{if}\ t_1 = \mathbf{set}\langle t'_1\rangle \land t_2 = \mathbf{set}\langle t'_2\rangle \\ \mathbf{map}\langle t'_1 \sqcup t'_2, t''_1 \sqcup t''_2\rangle & \mathbf{if}\ t_1 = \mathbf{map}\langle t'_1, t''_1\rangle \land t_2 = \mathbf{map}\langle t'_2, t''_2\rangle \\ \mathbf{value} & \mathbf{otherwise} \end{cases}$$

Expression Evaluation

The main judgment for Rascal Light expressions has the form $\boxed{e; \sigma \underset{\text{expr}}{\Longrightarrow} vres; \sigma'}$, where the expression e is evaluated in an initial store $\sigma \in \text{Var} \rightarrow \text{Val}$—mapping variables to values—returning a result *vres* and updated store σ' as a side-effect. The result *vres* is either an ordinary **success** v signifying successful execution or an exceptional result *exres*.

$$vres ::= \mathbf{success}\ v \mid exres$$

An exceptional result is either a control operation (**break**, **continue**, **fail**), an **error** that happened during execution, a thrown exception **throw** v or an early return value **return** v. The difference between **success** v and **return** v is that the latter should propagate directly from sub-expressions to the surrounding function call boundary, while the value in the former can be used further in intermediate computations (Example 6.4).

$$exres ::= \mathbf{return}\ v \mid \mathbf{throw}\ v \mid \mathbf{break} \mid \mathbf{continue} \mid \mathbf{fail} \mid \mathbf{error}$$

Example 6.4 (Early Return). The function that calculates producs, uses the early return functionality to short-circuit the rest of the calculation when a factor in the provided list is zero. If this branch is hit during execution, evaluating the expression produces **return** 0 as result, which is then propagated directly to function call boundary, skipping the rest of the loop and sequenced expressions.

```
int prod(list[int] xs) {
  int res = 1;
  for (x <- xs) {
    if (x == 0) return 0;
    else res *= x;
  };
  res
}
```

▲

Basic values evaluate to their semantic equivalent result values without any side effect (E-Val).

$$\text{E-VAL}\;\frac{}{vb;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\; vb;\sigma}$$

A variable evaluates to the value it is assigned to in the store if available (E-Var-Sucs), and otherwise result in an error (E-Var-Err).

$$\text{E-VAR-SUCS}\;\frac{x \in \operatorname{dom}\sigma}{x;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\;\sigma(x);\sigma} \qquad \text{E-VAR-ERR}\;\frac{x \notin \operatorname{dom}\sigma}{x;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma}$$

Unary expressions evaluate their operands, applying possible side-effects; if successful the corresponding semantic unary operator $[\![\ominus]\!]$ is applied on the result value (E-Un-Sucs), and otherwise it propagates the exceptional result (E-Un-Exc).

$$\text{E-UN-SUCS}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\; v;\sigma'}{\ominus\, e;\sigma \underset{\text{expr}}{\Longrightarrow} [\![\ominus]\!](v);\sigma'} \qquad \text{E-UN-EXC}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}{\ominus\, e;\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}$$

Example 6.5 (Unary operator semantics). $[\![-]\!](3)$ will evaluate to **success** -3, while $[\![-]\!](\{\})$ will evaluate to **error** ▲

Evaluating binary expressions is similar to unary expressions, requiring both operands to evaluate successfully (E-Bin-Sucs) to apply the corresponding semantic binary operator $[\![\oplus]\!]$; otherwise the exceptional results of the operands are propagated in order from left (E-Bin-Exc1) to right (E-Bin-Exc2).

$$\textsc{E-Bin-Sucs}\ \frac{e_1;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v_1;\sigma'' \qquad e_2;\sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v_2;\sigma'}{e_1 \oplus e_2;\sigma \underset{\text{expr}}{\Longrightarrow} [\![\oplus]\!](v_1, v_2);\sigma'}$$

$$\textsc{E-Bin-Exc1}\ \frac{e_1;\sigma \underset{\text{expr}}{\Longrightarrow} exres_1;\sigma''}{e_1 \oplus e_2;\sigma \underset{\text{expr}}{\Longrightarrow} exres_1;\sigma'}$$

$$\textsc{E-Bin-Exc2}\ \frac{e_1;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v_1;\sigma'' \qquad e_2;\sigma'' \underset{\text{expr}}{\Longrightarrow} exres_2;\sigma'}{e_1 \oplus e_2;\sigma \underset{\text{expr}}{\Longrightarrow} exres_2;\sigma'}$$

Constructor expressions evaluate their arguments first, and if they all successfully evaluate to values, then check whether the types of values match those expected in the declaration. If the result values have the right types and are not ■, a constructor value is constructed (E-Cons-Sucs), and otherwise a (type) error is produced (E-Cons-Err). In case any of the arguments has an exceptional result the evaluation of the rest of the arguments halts and the exceptional result is propagated (E-Cons-Exc).

$$\textsc{E-Cons-Sucs}\ \frac{\begin{array}{c}\textbf{data}\ at = \ldots \mid k(\underline{t}) \mid \ldots \qquad \underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \\ \underline{v : t'} \qquad\qquad \underline{v \neq \blacksquare} \qquad\qquad \underline{t' <: t}\end{array}}{k(\underline{e});\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ k(\underline{v});\sigma'}$$

$$\textsc{E-Cons-Err}\ \frac{\begin{array}{c}\textbf{data}\ at = \ldots \mid k(\underline{t}) \mid \ldots \qquad \underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \\ \underline{v : t'} \qquad\qquad \exists i. v_i = \blacksquare \vee t'_i \not<: t_i\end{array}}{k(\underline{e});\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma'}$$

$$\textsc{E-Cons-Exc}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} exres;\sigma'}{k(\underline{e});\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}$$

Evaluating list expressions also requires evaluating all subexpressions to a series of values; because of sequencing and necessity of early propagation of exceptional results, evaluation of series of subexpressions is done using a mutually recursive sequence evaluation judgment (see page 115). If the evaluation is successful then a list value is constructed

(E-List-Sucs), unless any value is undefined (■) in which case we produce an error (E-List-Err), and otherwise the exceptional result is propagated (E-List-Exc).

$$\textsc{E-List-Sucs}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \qquad \underline{v \neq \blacksquare}}{[\underline{e}];\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ [\underline{v}];\sigma'}$$

$$\textsc{E-List-Err}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \qquad \exists i.v_i = \blacksquare}{[\underline{e}];\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma'} \qquad \textsc{E-List-Exc}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} exres;\sigma'}{[\underline{e}];\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}$$

Set expression evaluation mirror the one for lists, except that values are constructed using a set constructor (E-Set-Sucs), which may reorder values and ensures that there are no duplicates. If any contained value was undefined (■) then an error is produced instead (E-Set-Err), and exceptional results are propagated (E-Set-Exc).

$$\textsc{E-Set-Sucs}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \qquad \underline{v \neq \blacksquare}}{\{\underline{e}\};\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ \{\underline{v}\};\sigma'}$$

$$\textsc{E-Set-Err}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v};\sigma' \qquad \exists i.v_i = \blacksquare}{\{\underline{e}\};\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma'} \qquad \textsc{E-Set-Exc}\ \frac{\underline{e};\sigma \underset{\text{expr}\star}{\Longrightarrow} exres;\sigma'}{\{\underline{e}\};\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}$$

Map expressions evaluate their keys and values in the declaration sequence, and if successful construct a map (E-Map-Sucs). Similarly to other collection expressions, errors are produced if any value is undefined (E-Map-Err) and exceptional results are propagated (E-Map-Exc).

$$\textsc{E-Map-Sucs}\ \frac{\underline{e,e'};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v,v'};\sigma' \qquad \underline{v \neq \blacksquare} \qquad \underline{v' \neq \blacksquare}}{(\underline{e : e'});\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\underline{v : v'});\sigma'}$$

$$\textsc{E-Map-Err}\ \frac{\underline{e,e'};\sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v,v'};\sigma' \qquad \exists .v_i = \blacksquare \lor v'_i = \blacksquare}{(\underline{e : e'});\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma'}$$

$$\textsc{E-Map-Exc}\ \frac{\underline{e,e'};\sigma \underset{\text{expr}\star}{\Longrightarrow} exres;\sigma'}{(\underline{e : e'});\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}$$

Lookup expressions require evaluating the outer expression to a map—otherwise producing an error (E-Lookup-Err)—and the index expression to a value. If the index is an existing key then the corresponding value is produced as result (E-Lookup-Sucs) and otherwise the nokey exception is thrown (E-Lookup-NoKey); here, I assume that

$$\textbf{data}\ \text{NoKey} = \text{nokey}(\text{value key})$$

is a built-in data-type definition. Exceptional results are propagated from sub-terms (E-Lookup-Exc1, E-Lookup-Exc2).

$$\textsc{E-Lookup-Sucs}\ \frac{\begin{array}{c} e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\ldots, v : v', \ldots); \sigma'' \\ e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \end{array}}{e_1[e_2]; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v'; \sigma'}$$

$$\textsc{E-Lookup-NoKey}\ \frac{\begin{array}{c} e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\underline{v : v'}); \sigma'' \\ e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v''; \sigma' \qquad \forall i. v'' \neq v_i \end{array}}{e_1[e_2]; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{throw}\ \text{nokey}(v''); \sigma'}$$

$$\textsc{E-Lookup-Err}\ \frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \qquad v \neq (\underline{v' : v''})}{e_1[e_2]; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error}; \sigma'}$$

$$\textsc{E-Lookup-Exc1}\ \frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{e_1[e_2]; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

$$\textsc{E-Lookup-Exc2}\ \frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\underline{v : v'}); \sigma'' \qquad e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{e_1[e_2]; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

Map update expressions also require the outer expression to evaluate to a map—otherwise producing an error (E-Update-Err1)—and the index and target expressions to evaluate to values. On succesful evaluation of both index and target value the map is updated, overriding the old value of the corresponding index if necessary (E-Update-Sucs) unless the index or target value is equal to ■ in which case an error is produced (E-Update-Err2). Finally, exceptional results are propagated left-to-right if necessary (E-Update-Exc1, E-Update-Exc2, E-Update-Exc3).

$$\textsc{E-Update-Sucs}\ \frac{\begin{array}{c} e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\underline{v : v'}); \sigma''' \qquad e_2; \sigma''' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v''; \sigma'' \\ e_3; \sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v'''; \sigma' \qquad v'' \neq \blacksquare \qquad v''' \neq \blacksquare \end{array}}{e_1[e_2 = e_3]; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ (\underline{v : v'}, v'' : v'''); \sigma'}$$

$$\textsc{E-Update-Err1}\ \frac{e_1;\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v;\sigma' \qquad v \neq (\underline{v' : v''})}{e_1[e_2 = e_3];\sigma \xRightarrow[\text{expr}]{} \textbf{error};\sigma'}$$

$$\textsc{E-Update-Err2}\ \frac{\begin{array}{cc} e_1;\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ (\underline{v : v'});\sigma''' & e_2;\sigma''' \xRightarrow[\text{expr}]{} \textbf{success}\ v'';\sigma'' \\ e_3;\sigma'' \xRightarrow[\text{expr}]{} \textbf{success}\ v''';\sigma' & v'' = \blacksquare \vee v''' = \blacksquare \end{array}}{e_1[e_2 = e_3];\sigma \xRightarrow[\text{expr}]{} \textbf{error};\sigma'}$$

$$\textsc{E-Update-Exc1}\ \frac{e_1;\sigma \xRightarrow[\text{expr}]{} \mathit{exres};\sigma'}{e_1[e_2 = e_3];\sigma \xRightarrow[\text{expr}]{} \mathit{exres};\sigma'}$$

$$\textsc{E-Update-Exc2}\ \frac{e_1;\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ (\underline{v : v'});\sigma'' \qquad e_2;\sigma'' \xRightarrow[\text{expr}]{} \mathit{exres};\sigma'}{e_1[e_2 = e_3];\sigma \xRightarrow[\text{expr}]{} \mathit{exres};\sigma'}$$

$$\textsc{E-Update-Exc3}\ \frac{\begin{array}{c} e_1;\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ (\underline{v : v'});\sigma'' \\ e_2;\sigma''' \xRightarrow[\text{expr}]{} \textbf{success}\ v'';\sigma'' \qquad e_3;\sigma'' \xRightarrow[\text{expr}]{} \mathit{exres};\sigma' \end{array}}{e_1[e_2 = e_3];\sigma \xRightarrow[\text{expr}]{} \mathit{exres};\sigma'}$$

Evaluation of function calls is more elaborate. Function definitions are statically scoped, and the semantics is eager, so arguments are evaluated using call-by-value. The initial step is thus to evaluate all the arguments to values if possible—propagating the exceptional result otherwise (E-Call-Arg-Exc)—and then check whether the values have the right type (otherwise producing an error, E-Call-Arg-Err). The evaluation proceeds by evaluating the body of the function with a fresh store that contains the values of global variables and the parameters bound to their respective argument values. There are then four cases:

1. If the body successfully evaluates to a correctly typed value, then that value is provided as the result (E-Call-Sucs).

2. If the body evaluates to a value that does not have the expected type, then it produces an error (E-Call-Res-Err1).

3. If the result is a thrown exception or error, then it is propagated (E-Call-Res-Exc).

4. Otherwise if the result is a control operator, then an error is produced (E-Call-Res-Err2).

In all cases the resulting store of executing the body is discarded—since local assignments fall out of scope—except the global variable values which are added to the store that was there before the function call.

$$\textsc{E-Call-Sucs}\ \frac{\begin{array}{c}\underline{\textbf{global}\ t_y\ y} \qquad \textbf{fun}\ t'\ f(\underline{t\ x}) = e' \\ \underline{e};\sigma \xRightarrow[\text{expr}\star]{} \textbf{success}\ \underline{v};\sigma'' \qquad \underline{v : t''} \qquad \underline{t'' <: t} \\ [\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v}]; e' \xRightarrow[\text{expr}]{} vres;\sigma' \\ vres = \textbf{return}\ v' \lor vres = \textbf{success}\ v' \qquad v' : t''' \qquad t''' <: t'\end{array}}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v';\sigma''[\underline{y \mapsto \sigma'(y)}]}$$

$$\textsc{E-Call-Arg-Err}\ \frac{\begin{array}{c}\textbf{fun}\ t'\ f(\underline{t\ x}) = e' \qquad \underline{e};\sigma \xRightarrow[\text{expr}\star]{} \textbf{success}\ \underline{v};\sigma' \\ \underline{v : t''} \qquad t''_i \not<: t_i\end{array}}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} \textbf{error};\sigma'}$$

$$\textsc{E-Call-Arg-Exc}\ \frac{\underline{e};\sigma \xRightarrow[\text{expr}\star]{} exres;\sigma'}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} exres;\sigma'}$$

$$\textsc{E-Call-Res-Exc}\ \frac{\begin{array}{c}\underline{\textbf{global}\ t_y\ y} \qquad \textbf{fun}\ t'\ f(\underline{t\ x}) = e' \\ \underline{e};\sigma \xRightarrow[\text{expr}\star]{} \textbf{success}\ \underline{v};\sigma'' \qquad \underline{v : t''} \qquad \underline{t'' <: t} \\ [\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v}]; e' \xRightarrow[\text{expr}]{} exres;\sigma' \qquad exres = \textbf{throw}\ v'\end{array}}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} exres;\sigma''[\underline{y \mapsto \sigma'(y)}]}$$

$$\textsc{E-Call-Res-Err1}\ \frac{\begin{array}{c}\underline{\textbf{global}\ t_y\ y} \qquad \textbf{fun}\ t'\ f(\underline{t\ x}) = e' \\ \underline{e};\sigma \xRightarrow[\text{expr}\star]{} \textbf{success}\ \underline{v};\sigma'' \qquad \underline{v : t''} \qquad \underline{t'' <: t} \\ [\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v}]; e' \xRightarrow[\text{expr}]{} vres;\sigma' \\ vres = \textbf{return}\ v' \lor vres = \textbf{success}\ v' \qquad v' : t''' \qquad t''' \not<: t'\end{array}}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} \textbf{error};\sigma''[\underline{y \mapsto \sigma'(y)}]}$$

$$\textsc{E-Call-Res-Err2}\ \frac{\begin{array}{c}\underline{\textbf{global}\ t_y\ y} \qquad \textbf{fun}\ t'\ f(\underline{t\ x}) = e' \\ \underline{e};\sigma \xRightarrow[\text{expr}\star]{} \textbf{success}\ \underline{v};\sigma'' \qquad \underline{v : t''} \qquad \underline{t'' <: t} \\ [\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v}]; e' \xRightarrow[\text{expr}]{} exres;\sigma' \\ exres \in \{\textbf{break}, \textbf{continue}, \textbf{fail}, \textbf{error}\}\end{array}}{f(\underline{e});\sigma \xRightarrow[\text{expr}]{} \textbf{error};\sigma''[\underline{y \mapsto \sigma'(y)}]}$$

The **return**-expression evaluates its argument expression first, and if it successfully produces a value then the result would be an early return with that value (E-Ret-Sucs); recall that early returns are treated as exceptional values and so propagated through most evaluation rules, except at function call boundaries (rules E-Call-Sucs and E-Call-Res-Err1). Otherwise, the exceptional result is propagated (E-Ret-Exc).

$$\textsc{E-Ret-Sucs}\ \frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v;\sigma'}{\textbf{return}\ e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{return}\ v;\sigma'} \qquad \textsc{E-Ret-Exc}\ \frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \mathit{exres};\sigma'}{\textbf{return}\ e;\sigma \underset{\text{expr}}{\Longrightarrow} \mathit{exres};\sigma'}$$

In Rascal Light, variables must be declared before being assigned, and declarations are unique since shadowing is disallowed. Evaluating an assignment proceeds by evaluating the right-hand side expression—propagating exceptional results (E-Asgn-Exc)—and then checking whether the produced value is compatible with the declared type. If it is compatible then the store is updated (E-Asgn-Sucs), and otherwise an error is produced (E-Asgn-Err).

$$\textsc{E-Asgn-Sucs}\ \frac{\begin{array}{cc}\textbf{local}\ t\ x \vee \textbf{global}\ t\ x & e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v;\sigma' \\ v : t' & t' <: t\end{array}}{x = e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v;\sigma'[x \mapsto v]}$$

$$\textsc{E-Asgn-Err}\ \frac{\begin{array}{cc}\textbf{local}\ t\ x \vee \textbf{global}\ t\ x & e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v;\sigma' \\ v : t' & t' \not<: t\end{array}}{x = e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error};\sigma'}$$

$$\textsc{E-Asgn-Exc}\ \frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \mathit{exres};\sigma'}{x = e;\sigma \underset{\text{expr}}{\Longrightarrow} \mathit{exres};\sigma'}$$

The **if**-expression works like other languages: the **then**-branch is evaluated if the condition is true (E-If-True), otherwise the **else**-branch is evaluated (E-If-False). If the conditional produces a non-Boolean value then an error is raised (E-If-Err) and otherwise exceptional results are propagated (E-If-Exc).

$$\textsc{E-If-True}\ \frac{e_{\text{cond}};\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ \text{true}();\sigma'' \qquad e_1;\sigma'' \underset{\text{expr}}{\Longrightarrow} \mathit{vres}_1;\sigma'}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\sigma \underset{\text{expr}}{\Longrightarrow} \mathit{vres}_1;\sigma'}$$

$$\textsc{E-If-False}\ \frac{e_{\text{cond}}; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ \text{false}(); \sigma'' \qquad e_2; \sigma'' \xRightarrow[\text{expr}]{} \mathit{vres}_2; \sigma'}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2; \sigma \xRightarrow[\text{expr}]{} \mathit{vres}_2; \sigma'}$$

$$\textsc{E-If-Err}\ \frac{e_{\text{cond}}; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v; \sigma' \qquad v \neq \text{true}() \qquad v \neq \text{false}()}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2; \sigma \xRightarrow[\text{expr}]{} \textbf{error}; \sigma'}$$

$$\textsc{E-If-Exc}\ \frac{e_{\text{cond}}; \sigma \xRightarrow[\text{expr}]{} \mathit{exres}; \sigma'}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2; \sigma \xRightarrow[\text{expr}]{} \mathit{exres}; \sigma'}$$

The **switch**-expression initially evaluates the scrutinee expression (e), and then proceeds to execute the cases (discussed on page 116) on the result value (E-SWITCH-SUCS). The evaluation of cases is allowed to **fail**, in which case the evaluation is successful and has the special value $\blacksquare$ (E-SWITCH-FAIL); other exceptional results are propagated as usual (E-SWITCH-EXC1, E-SWITCH-EXC2).

$$\textsc{E-Switch-Sucs}\ \frac{e; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{cases}]{} \textbf{success}\ v'; \sigma'}{\textbf{switch}\ e\ \underline{cs}; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v'; \sigma'}$$

$$\textsc{E-Switch-Fail}\ \frac{e; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{cases}]{} \textbf{fail}; \sigma'}{\textbf{switch}\ e\ \underline{cs}; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ \blacksquare; \sigma'}$$

$$\textsc{E-Switch-Exc1}\ \frac{e; \sigma \xRightarrow[\text{expr}]{} \mathit{exres}; \sigma'}{\textbf{switch}\ e\ \underline{cs}; \sigma \xRightarrow[\text{expr}]{} \mathit{exres}; \sigma'}$$

$$\textsc{E-Switch-Exc2}\ \frac{e; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{cases}]{} \mathit{exres}; \sigma' \qquad \mathit{exres} \neq \textbf{fail}}{\textbf{switch}\ e\ \underline{cs}; \sigma \xRightarrow[\text{expr}]{} \mathit{exres}; \sigma'}$$

The **visit**-expression has similar evaluation cases to **switch** (E-VISIT-SUCS, E-VISIT-FAIL, E-VISIT-EXC1, E-VISIT-EXC2), except that the cases are evaluated using the visit relation that traverses the produced value of target expression given the provided strategy.

$$\textsc{E-Visit-Sucs}\ \dfrac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{visit}]{st} \textbf{success}\ v'; \sigma'}{st\ \textbf{visit}\ e\ \underline{cs}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v'; \sigma'}$$

$$\textsc{E-Visit-Fail}\ \dfrac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{visit}]{st} \textbf{fail}; \sigma'}{st\ \textbf{visit}\ e\ \underline{cs}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'}$$

$$\textsc{E-Visit-Exc1}\ \dfrac{e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{st\ \textbf{visit}\ e\ \underline{cs}; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

$$\textsc{E-Visit-Exc2}\ \dfrac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'' \qquad \underline{cs}; v; \sigma'' \xRightarrow[\text{visit}]{st} exres; \sigma' \qquad exres \neq \textbf{fail}}{st\ \textbf{visit}\ e\ \underline{cs}; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

The control operations **break**, **continue** and **fail** evaluate to themselves without any side-effects (E-Break, E-Continue, E-Fail).

$$\textsc{E-Break}\ \dfrac{}{\textbf{break}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{break}; \sigma} \qquad \textsc{E-Fail}\ \dfrac{}{\textbf{fail}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{fail}; \sigma}$$

$$\textsc{E-Continue}\ \dfrac{}{\textbf{continue}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{continue}; \sigma}$$

Blocks allow evaluating inner expressions using a local declaration of variables, which are then afterwards removed from the resulting store (E-Block-Sucs, E-Block-Exc). Recall, that we consider an implicit definition environment based on scoping, and so the local declarations in the block will be implicitly available in the evaluation of the body subexpression sequence.

$$\textsc{E-Block-Sucs}\ \dfrac{\underline{e}; \sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v}; \sigma'}{\textbf{local}\ \underline{t\ x}\ \textbf{in}\ \underline{e}\ \textbf{end}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ \text{last}(\underline{v}); (\sigma' \setminus \underline{x})}$$

$$\textsc{E-Block-Exc}\ \dfrac{\underline{e}; \sigma \underset{\text{expr}\star}{\Longrightarrow} exres; \sigma'}{\textbf{local}\ \underline{t\ x}\ \textbf{in}\ \underline{e}\ \textbf{end}; \sigma \underset{\text{expr}}{\Longrightarrow} exres; (\sigma' \setminus \underline{x})}$$

The auxiliary function last is here used to extract the last element in the sequence (or return ■ if empty).

$$\begin{aligned}\mathrm{last}(v_1, \ldots, v_\mathrm{n}, v') &= v' \\ \mathrm{last}(\varepsilon) &= \blacksquare\end{aligned}$$

The **for**-loop evaluates target generator expression to a set of possible *environments* that represent possible assignments of variables to values—propagating exceptions if necessary (E-For-Exc)—and then it iterates over each possible assignment using the each-relation (E-For-Sucs).

$$\textsc{E-For-Sucs}\,\frac{g; \sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{success } \underline{\rho}; \sigma'' \quad e; \underline{\rho}; \sigma'' \underset{\text{each}}{\Longrightarrow} vres; \sigma'}{\textbf{for } g\ e; \sigma \underset{\text{expr}}{\Longrightarrow} vres; \sigma'} \qquad \textsc{E-For-Exc}\,\frac{g; \sigma \underset{\text{gexpr}}{\Longrightarrow} exres; \sigma'}{\textbf{for } g\ e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

The evaluation of **while**-loops is analogous to other imperative languages with control operations, in that the body of the while loop is continuously executed until the target condition does not hold (E-While-False). If the body successfully finishes with an value or **continue** then iteration continues (E-While-True-Sucs), if the body finishes with **break** the iteration stops with value ■ (E-While-True-Break), if the conditional evaluates to a non-Boolean value it errors out (E-While-Err) and otherwise if another kind of exceptional result is produced then it is propagated (E-While-Exc1, E-While-Exc2).

$$\textsc{E-While-False}\,\frac{e_\text{cond}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success } \mathrm{false}(); \sigma'}{\textbf{while } e_\text{cond}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success } \blacksquare; \sigma'}$$

$$\textsc{E-While-True-Sucs}\,\frac{\begin{array}{c} e_\text{cond}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success } \mathrm{true}(); \sigma'' \qquad e; \sigma'' \underset{\text{expr}}{\Longrightarrow} vres; \sigma''' \\ vres = \textbf{success } v \lor vres = \textbf{continue} \\ \textbf{while } e_\text{cond}\ e; \sigma''' \underset{\text{expr}}{\Longrightarrow} vres'; \sigma' \end{array}}{\textbf{while } e_\text{cond}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} vres'; \sigma'}$$

$$\textsc{E-While-True-Break}\,\frac{e_\text{cond}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success } \mathrm{true}(); \sigma'' \qquad e; \sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{break}; \sigma'}{\textbf{while } e_\text{cond}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success } \blacksquare; \sigma'}$$

$$\text{E-WHILE-EXC1}\ \frac{e_{\text{cond}}; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{\textbf{while}\ e_{\text{cond}}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

$$\text{E-WHILE-EXC2}\ \frac{\begin{array}{c} e_{\text{cond}}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ \text{true}(); \sigma'' \qquad e; \sigma'' \underset{\text{expr}}{\Longrightarrow} exres; \sigma' \\ exres \in \{\textbf{throw}\ v, \textbf{return}\ v, \textbf{fail}, \textbf{error}\} \end{array}}{\textbf{while}\ e_{\text{cond}}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

$$\text{E-WHILE-ERR}\ \frac{e_{\text{cond}}; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \qquad v \neq \text{true}() \qquad v \neq \text{false}()}{\textbf{while}\ e_{\text{cond}}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error}; \sigma'}$$

The **solve**-loop keeps evaluating the body expression until the provided variables reach a fixed-point; this is analogous to the TRON **fix**-statement presented in Chapter 5. Initially, the body expression is evaluated and then the values of target variables is compared from before and after iteration; if the values are equal after an iteration, then evaluation stops (E-SOLVE-EQ) and otherwise the iteration continues (E-SOLVE-NEQ). If any of the variables do not have a value assigned, an error is produced (E-SOLVE-ERR), and otherwise if an exceptional result is produced, it is propagated (E-SOLVE-EXC).

$$\text{E-SOLVE-EQ}\ \frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \qquad \underline{x} \subseteq \text{dom}\ \sigma \cap \text{dom}\ \sigma' \qquad \underline{\sigma(x) = \sigma'(x)}}{\textbf{solve}\ \underline{x}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'}$$

$$\text{E-SOLVE-NEQ}\ \frac{\begin{array}{c} e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'' \qquad \underline{x} \subseteq \text{dom}\ \sigma \cap \text{dom}\ \sigma'' \\ \exists i. \sigma(x_i) \neq \sigma''(x_i) \qquad \textbf{solve}\ \underline{x}\ e; \sigma'' \underset{\text{expr}}{\Longrightarrow} vres; \sigma' \end{array}}{\textbf{solve}\ \underline{x}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} vres; \sigma'}$$

$$\text{E-SOLVE-EXC}\ \frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{\textbf{solve}\ \underline{x}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

$$\text{E-SOLVE-ERR}\ \frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \qquad x_i \notin \text{dom}\ \sigma \cap \text{dom}\ \sigma'}{\textbf{solve}\ \underline{x}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error}; \sigma'}$$

The **throw**-expression, evaluates its inner expression first—propagating exceptional results if necessary (E-Thr-Exc)—and then produces a **throw** result with result value (E-Thr-Sucs).

$$\textsc{E-Thr-Sucs}\,\frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'}{\textbf{throw}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{throw}\ v; \sigma'} \qquad \textsc{E-Thr-Exc}\,\frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{\textbf{throw}\ e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

The **try-finally** expression executes the **try**-body first and then the **finally**-body. If the **finally**-body produces an exceptional result during execution then that result is propagated (E-Fin-Exc) and otherwise the **try**-body result value is used (E-Fin-Sucs).

$$\textsc{E-Fin-Sucs}\,\frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} vres_1; \sigma'' \qquad e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v_2; \sigma'}{\textbf{try}\ e_1\ \textbf{finally}\ e_2; \sigma \underset{\text{expr}}{\Longrightarrow} vres_1; \sigma'}$$

$$\textsc{E-Fin-Exc}\,\frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} vres_1; \sigma'' \qquad e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow} exres_2; \sigma'}{\textbf{try}\ e_1\ \textbf{finally}\ e_2; \sigma \underset{\text{expr}}{\Longrightarrow} exres_2; \sigma'}$$

The **try-catch** expression evaluates the **try**-body and if it produces a thrown value, then it binds the value in the body of **catch** and continues evaluation (E-Try-Catch). For all other results, it simply propagates them without evaluating the **catch**-body (E-Try-Ord).

$$\textsc{E-Try-Catch}\,\frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{throw}\ v_1; \sigma'' \qquad e_2; \sigma''[x \mapsto v_1] \underset{\text{expr}}{\Longrightarrow} vres_2; \sigma'}{\textbf{try}\ e_1\ \textbf{catch}\ x \Rightarrow e_2; \sigma \underset{\text{expr}}{\Longrightarrow} vres_2; (\sigma' \setminus x)}$$

$$\textsc{E-Try-Ord}\,\frac{e_1; \sigma \underset{\text{expr}}{\Longrightarrow} vres_1; \sigma' \qquad vres_1 \neq \textbf{throw}\ v_1}{\textbf{try}\ e_1\ \textbf{catch}\ x \Rightarrow e_2; \sigma \underset{\text{expr}}{\Longrightarrow} vres_1; \sigma'}$$

Expression Sequences Evaluating a sequence of expressions proceeds by evaluating each expression, combining the results if successful (ES-Emp, ES-More) and otherwise propagating the first exceptional result encountered (ES-Exc1, ES-Exc2).

$$\textsc{ES-More}\,\frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma'' \qquad \underline{e'}; \sigma'' \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ \underline{v'}; \sigma'}{e, \underline{e'}; \sigma \underset{\text{expr}\star}{\Longrightarrow} \textbf{success}\ v, \underline{v'}; \sigma'}$$

$$\textsc{ES-Emp}\ \frac{}{\textbf{success}\ \varepsilon;\sigma \xRightarrow[\text{expr}\star]{} \varepsilon;\sigma} \qquad \textsc{ES-Exc1}\ \frac{e;\sigma \xRightarrow[\text{expr}]{} exres;\sigma'}{e,\underline{e'};\sigma \xRightarrow[\text{expr}\star]{} exres;\sigma'}$$

$$\textsc{ES-Exc2}\ \frac{e;\sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v;\sigma'' \qquad \underline{e'};\sigma'' \xRightarrow[\text{expr}\star]{} exres;\sigma'}{e,\underline{e'};\sigma \xRightarrow[\text{expr}\star]{} exres;\sigma'}$$

Cases The evaluation relation for evaluating a series of cases has the form $\boxed{\underline{cs};v;\sigma \xRightarrow[\text{cases}]{} vres;\sigma'}$, and intuitively proceeds by sequentially evaluating each case (in $\underline{cs}$) against value v until one of them produces a non-**fail** result. For each case, the first step is to match the given value against target pattern and then evaluate the target expression under the set of possible matches; if the evaluation of the target expression produces a **fail** as result, the rest of the cases are evaluated in a restored initial state (ECS-More-Fail) and otherwise the result is propagated (ECS-More-Ord). If all possible cases are exhausted, the result is **fail** (ECS-Emp).

$$\textsc{ECS-Emp}\ \frac{}{\varepsilon;v;\sigma \xRightarrow[\text{cases}]{} \textbf{fail};\sigma}$$

$$\textsc{ECS-More-Fail}\ \frac{\begin{array}{c}\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho} \qquad \underline{\rho};e;\sigma \xRightarrow[\text{case}]{} \textbf{fail};\sigma'' \\ \underline{cs};v;\sigma \xRightarrow[\text{cases}]{} vres;\sigma'\end{array}}{\textbf{case}\ p \Rightarrow e,\underline{cs};v;\sigma \xRightarrow[\text{cases}]{} vres;\sigma'}$$

$$\textsc{ECS-More-Ord}\ \frac{\begin{array}{c}\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho} \qquad \underline{\rho};e;\sigma \xRightarrow[\text{case}]{} vres;\sigma' \\ vres \neq \textbf{fail}\end{array}}{\textbf{case}\ p \Rightarrow e,\underline{cs};v;\sigma \xRightarrow[\text{cases}]{} vres;\sigma'}$$

Evaluating a single case—with relation $\boxed{\underline{\rho};e;\sigma \xRightarrow[\text{case}]{} vres;\sigma'}$—requires trying each possible binding (in $\underline{\rho}$) sequentially, producing **fail** if no binding is available (EC-Emp). If evaluating target expression produces a non-**fail** value then it is propagated (EC-More-Ord), otherwise the rest of the possible bindings are tried in a restored initial state (EC-More-Fail).

$$\textsc{EC-More-Fail}\ \frac{e; \sigma\rho \xRightarrow[\text{expr}]{} \mathbf{fail}; \sigma'' \qquad \underline{\rho'}; e; \sigma \xRightarrow[\text{case}]{} vres; \sigma'}{\rho, \underline{\rho'}; e; \sigma \xRightarrow[\text{case}]{} vres; \sigma'}$$

$$\textsc{EC-Emp}\ \frac{}{\varepsilon; e; \sigma \xRightarrow[\text{case}]{} \mathbf{fail}; \sigma} \qquad \textsc{EC-More-Ord}\ \frac{e; \sigma\rho \xRightarrow[\text{expr}]{} vres; \sigma' \qquad vres \neq \mathbf{fail}}{\rho, \underline{\rho'}; e; \sigma \xRightarrow[\text{case}]{} vres; (\sigma' \setminus \mathrm{dom}\ \rho)}$$

Traversals One of the key features of Rascal is **visit**-expressions which provide generic traversals over data values and collections, allowing for multiple strategies to determine the traversal ordering and halting conditions. In a traditional object-oriented language or functional language, transforming large structures is cumbersome and requires a great amount of boilerplate, requiring a function for each type of datatype, where each function must deconstruct the input data, applying target changes, recursively calling the right traversal functions for traversal of further contained data and reconstructing the data with new values. Precisely, the first-class handling of these aspects combined with the other features such as the powerful pattern matching makes Rascal particularly suitable as a high-level transformation language.

The main traversal relation $\boxed{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{st} vres; \sigma'}$ delegates execution to the correct strategy-dependent traversal, and performs fixed-point calculation if necessary. For the **top-down-break** and **top-down** strategies it uses the top-down traversal relation $\boxed{\underline{cs}; v; \sigma \xRightarrow[\text{td-visit}]{br} vres; \sigma'}$ specifying **break** and no **no-break** as breaking strategies respectively (EV-TD and EV-TDB); this works analogously with the **bottom-up-break** (EV-BU and EV-BUB) and **bottom-up** strategies using the $\boxed{\underline{cs}; v; \sigma \xRightarrow[\text{bu-visit}]{br} vres; \sigma'}$ relation.

$$\text{EV-TD}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td-visit}]{\textbf{no-break}} vres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{top-down}} vres; \sigma'} \qquad \text{EV-TDB}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td-visit}]{\textbf{break}} vres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{top-down-break}} vres; \sigma'}$$

$$\text{EV-BU}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{bu-visit}]{\textbf{no-break}} vres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{bottom-up}} vres; \sigma'} \qquad \text{EV-BUB}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{bu-visit}]{\textbf{break}} vres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{bottom-up-break}} vres; \sigma'}$$

The **innermost** streatgy evaluates the **bottom-up** traversal as long as it produces a resulting value not equal to the one from the previous iteration (EV-IM-Neq), returning the result value when a fixed-point is reached (EV-IM-Eq); if any exceptional result happens during evaluation it will be propagated (EV-IM-Exc). Analogous evaluation steps happens with **outermost** streatgy and **top-down** traversal (EV-OM-Neq, EV-OM-Eq, EV-OM-Exc).

$$\text{EV-IM-EQ}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{\textbf{no-break}} \textbf{success}\ v; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{innermost}} \textbf{success}\ v; \sigma'} \qquad \text{EV-IM-EXC}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{\textbf{no-break}} exres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{innermost}} exres; \sigma'}$$

$$\text{EV-IM-NEQ}\ \frac{\begin{array}{c}\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{\textbf{no-break}} \textbf{success}\ v'; \sigma'' \qquad v \neq v' \\ \underline{cs}; v'; \sigma'' \xRightarrow[\text{visit}]{\textbf{innermost}} vres; \sigma'\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{innermost}} vres; \sigma'}$$

$$\text{EV-OM-EQ}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{\textbf{no-break}} \textbf{success}\ v; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{outermost}} \textbf{success}\ v; \sigma'} \qquad \text{EV-OM-EXC}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{\textbf{no-break}} exres; \sigma'}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{outermost}} exres; \sigma'}$$

$$\text{EV-OM-NEQ}\ \frac{\begin{array}{c}\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{\textbf{no-break}} \textbf{success}\ v'; \sigma'' \qquad v \neq v' \\ \underline{cs}; v'; \sigma'' \xRightarrow[\text{visit}]{\textbf{outermost}} vres; \sigma'\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{\textbf{outermost}} vres; \sigma'}$$

The top-down traversal strategy starts by executing all cases on the target value, *scrutinee*, applying possible replacements and effects to produce an intermediate result value; the traversal then continues on the sequence of contained values of the this intermediate result, finally reconstructing a new output containing the possible replacement values obtained (ETV-Ord-Sucs1, ETV-Ord-Sucs2). If using the **break** strategy, the traversal will stop at the first value that produces a successful result (ETV-Break-Sucs); otherwise, if any sub-result produces a non-**fail** exceptional result it is propagated (ETV-Exc1, ETV-Exc2).

$$\text{ETV-BREAK-SUCS}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \textbf{success}\ v'; \sigma' \qquad br = \textbf{break}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \textbf{success}\ v'; \sigma'}$$

$$\textrm{ETV-}\textsc{Ord-Sucs1}\ \frac{\begin{array}{c}\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \mathit{vfres}; \sigma'' \qquad br = \mathbf{break} \Rightarrow \mathit{vfres} = \mathbf{fail} \\ v'' = \text{if-fail}(\mathit{vfres}, v) \qquad \underline{v'''} = \text{children}(v'') \\ \underline{cs}; \underline{v'''}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br} \mathbf{fail}; \sigma'\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathit{vfres}; \sigma'}$$

$$\textrm{ETV-}\textsc{Ord-Sucs2}\ \frac{\begin{array}{cc}\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \mathit{vfres}; \sigma'' & br = \mathbf{break} \Rightarrow \mathit{vfres} = \mathbf{fail} \\ v'' = \text{if-fail}(\mathit{vfres}, v) & \underline{v'''} = \text{children}(v'') \\ \underline{cs}; \underline{v'''}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br} \mathbf{success}\ \underline{v''''}; \sigma' & \mathbf{recons}\ v''\ \mathbf{using}\ \underline{v''''}\ \mathbf{to}\ \mathit{rcres}\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathit{rcres}; \sigma'}$$

$$\textrm{ETV-}\textsc{Exc1}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \mathit{exres}; \sigma' \qquad \mathit{exres} \neq \mathbf{fail}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathit{exres}; \sigma'}$$

$$\textrm{ETV-}\textsc{Exc2}\ \frac{\begin{array}{cc}\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \mathit{vfres}; \sigma'' & br = \mathbf{break} \Rightarrow \mathit{vfres} = \mathbf{fail} \\ v'' = \text{if-fail}(\mathit{vfres}, v) & \underline{v'''} = \text{children}(v'') \\ \underline{cs}; \underline{v'''}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br} \mathit{exres}; \sigma' & \mathit{exres} \neq \mathbf{fail}\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathit{exres}; \sigma'}$$

Evaluating a sequence of top-down traversals, requires executing a top-down traversal for each element, failing if the input sequence is empty (ETVS-Emp) and otherwise combining the results (ETVS-More). If the **break** strategy is used, then the iteration will instead stop at first succesful result (ETVS-Break), and any non-**fail** exceptional result is propagated (ETVS-Exc1, ETVS-Exc2).

$$\textrm{ETVS-}\textsc{Emp}\ \frac{}{\underline{cs}; \varepsilon; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} \mathbf{fail}; \sigma}$$

$$\textrm{ETVS-}\textsc{Break}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathbf{success}\ v''; \sigma' \qquad br = \mathbf{break}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} \mathbf{success}\ v'', \underline{v'}; \sigma'}$$

$$\textrm{ETVS-}\textsc{More}\ \frac{\begin{array}{c}\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} \mathit{vfres}; \sigma'' \qquad br = \mathbf{break} \Rightarrow \mathit{vfres} = \mathbf{fail} \\ \underline{cs}; \underline{v'}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br} \mathit{vfres}\star'; \sigma'\end{array}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} \text{vcombine}(\mathit{vfres}, \mathit{vfres}\star', v, \underline{v'}); \sigma'}$$

$$\text{ETVS-Exc1}\ \frac{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} exres; \sigma' \qquad exres \neq \textbf{fail}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} exres; \sigma'}$$

$$\text{ETVS-Exc2}\ \frac{\begin{array}{cc} \underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} vfres; \sigma'' & br = \textbf{break} \Rightarrow vfres = \textbf{fail} \\ \underline{cs}; \underline{v'}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br} exres; \sigma' & exres \neq \textbf{fail} \end{array}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} exres; \sigma'}$$

Bottom-up traversals work analogously to top-down traversals, except that traversal of children and reconstruction happens before traversing the final reconstructed value (EBU-Break-Sucs, EBU-No-Break-Sucs, EBU-Fail-Sucs). The analogy also holds with propagation of exceptional results and errors (EBU-Exc, EBU-No-Break-Err, EBU-No-Break-Exc).

$$\text{EBU-Break-Sucs}\ \frac{\begin{array}{cc} \underline{v''} = \text{children}(v) & \underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{success}\ \underline{v'}; \sigma' \\ br = \textbf{break} & \textbf{recons}\ v\ \textbf{using}\ \underline{v'}\ \textbf{to}\ rcres \end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} rcres; \sigma'}$$

$$\text{EBU-No-Break-Sucs}\ \frac{\begin{array}{cc} \underline{v''} = \text{children}(v) & \underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{success}\ \underline{v'''}; \sigma'' \\ br = \textbf{no-break} & \textbf{recons}\ v\ \textbf{using}\ \underline{v'''}\ \textbf{to success}\ v' \\ \multicolumn{2}{c}{\underline{cs}; v'; \sigma'' \xRightarrow[\text{cases}]{} vfres'; \sigma'} \end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \textbf{success}\ \text{if-fail}(vfres', v'); \sigma'}$$

$$\text{EBU-Fail-Sucs}\ \frac{\begin{array}{cc} \underline{v''} = \text{children}(v) & \underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{fail}; \sigma'' \\ \multicolumn{2}{c}{\underline{cs}; v; \sigma'' \xRightarrow[\text{cases}]{} vres; \sigma'} \end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} vres; \sigma'}$$

$$\text{EBU-Exc}\ \frac{\underline{v'} = \text{children}(v) \qquad \underline{cs}; \underline{v'}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} exres; \sigma' \qquad exres \neq \textbf{fail}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} exres; \sigma'}$$

$$\textsc{EBU-No-Break-Err}\ \dfrac{\begin{array}{cc}\underline{v''} = \text{children}(v) & \underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{success}\ \underline{v'''}; \sigma' \\ br = \textbf{no-break} & \textbf{recons}\ v\ \textbf{using}\ \underline{v'''}\ \textbf{to error}\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \textbf{error}; \sigma'}$$

$$\textsc{EBU-No-Break-Exc}\ \dfrac{\begin{array}{cc}\underline{v''} = \text{children}(v) & \underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{success}\ \underline{v'''}; \sigma'' \\ br = \textbf{no-break} & \textbf{recons}\ v\ \textbf{using}\ \underline{v'''}\ \textbf{to success}\ v' \\ \multicolumn{2}{c}{\underline{cs}; v'; \sigma'' \xRightarrow[\text{cases}]{} \mathit{exres}; \sigma'}\end{array}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \mathit{exres}; \sigma'}$$

Evaluating a sequence of bottom-up traversals is analogous to evaluating a sequence of top-down traversals. Each element in the sequence is evaluated and their results is combined (EBUS-Emp,EBUS-More), stopping at the first succesful result when using the break strategy (EBUS-Break). Otherwise, non-**fail** exceptional results are propagated (EBUS-Exc1, EBUS-Exc2).

$$\textsc{EBUS-Emp}\ \dfrac{}{\underline{cs}; \varepsilon; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{fail}; \sigma}$$

$$\textsc{EBUS-Break}\ \dfrac{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \textbf{success}\ v''; \sigma \qquad br = \textbf{break}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \textbf{success}\ v'', \underline{v'}; \sigma'}$$

$$\textsc{EBUS-More}\ \dfrac{\begin{array}{c}\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \mathit{vfres}; \sigma'' \qquad br = \textbf{break} \Rightarrow \mathit{vfres} = \textbf{fail} \\ \underline{cs}; \underline{v'}; \sigma'' \xRightarrow[\text{bu}-\text{visit}\star]{br} \mathit{vfres}\star'; \sigma'\end{array}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \text{vcombine}(\mathit{vfres}, \mathit{vfres}\star', v, \underline{v'}); \sigma'}$$

$$\textsc{EBUS-Exc1}\ \dfrac{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \mathit{exres}; \sigma' \qquad \mathit{exres} \neq \textbf{fail}}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \mathit{exres}; \sigma'}$$

$$\text{EBUS-Exc2}\;\frac{\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br} \mathit{vfres}; \sigma'' \quad br = \textbf{break} \Rightarrow \mathit{vfres} = \textbf{fail} \quad \underline{cs}; \underline{v'}; \sigma'' \xRightarrow[\text{bu}-\text{visit}\star]{br} \mathit{exres}; \sigma'}{\underline{cs}; v, \underline{v'}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br} \mathit{exres}; \sigma'}$$

Auxiliary The children function extracts the directly contained values of the given input value.

$$\text{children}(vb) = \varepsilon \quad \text{children}(k(\underline{v})) = \underline{v}$$
$$\text{children}([\underline{v}]) = \underline{v} \quad \text{children}(\{\underline{v}\}) = \underline{v}$$
$$\text{children}((\underline{v : v'})) = \underline{v}, \underline{v'} \quad \text{children}(\blacksquare) = \varepsilon$$

$$\mathit{vfres} ::= \textbf{success}\ v \mid \textbf{fail}$$

The if-fail function will return a provided default value if the first argument is **fail**, and otherwise it will use the provided value in the first argument.

$$\text{if-fail}(\textbf{fail}, v) = v \quad \text{if-fail}(\textbf{success}\ v', v) = v'$$

The vcombine function will combine **success** and **fail** results from visitor, producing **fail** if both result arguments are **fail** otherwise producing **success** result, possibly using default values

$$\text{vcombine}(\mathit{vfres}, \mathit{vfres}\star', v, \underline{v'}) = \begin{cases} \textbf{fail} & \textbf{if} \begin{array}{r} \mathit{vfres} = \textbf{fail} \land \\ \mathit{vfres}' = \textbf{fail} \end{array} \\ \textbf{success} \begin{pmatrix} \text{if-fail}(\mathit{vfres}, v), \\ \text{if-fail}(\mathit{vfres}\star', \underline{v'}) \end{pmatrix} & \textbf{otherwise} \end{cases}$$

The reconstruction relation $\boxed{\textbf{recons}\ v\ \textbf{using}\ \underline{v'}\ \textbf{to}\ \mathit{rcres}}$ tries to update the elements of a value, checking whether provided values are type correct and defined (not ■), otherwise producing an error.

$$\mathit{rcres} ::= \textbf{success}\ v \mid \textbf{error}$$

$$\textsc{RC-Val-Sucs}\ \frac{}{\mathbf{recons}\ vb\ \mathbf{using}\ \varepsilon\ \mathbf{to\ success}\ vb}$$

$$\textsc{RC-Val-Err}\ \frac{}{\mathbf{recons}\ vb\ \mathbf{using}\ v', \underline{v''}\ \mathbf{to\ error}}$$

$$\textsc{RC-Cons-Sucs}\ \frac{\mathbf{data}\ at = \ldots \mid k(\underline{t}) \mid \ldots \qquad \underline{v' : t'} \qquad \underline{v' \neq \blacksquare} \qquad \underline{t' <: t}}{\mathbf{recons}\ k(\underline{v})\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ success}\ k(\underline{v'})}$$

$$\textsc{RC-Cons-Err}\ \frac{\mathbf{data}\ at = \ldots \mid k(\underline{t}) \mid \ldots \qquad \underline{v' : t'} \qquad v_i = \blacksquare \vee t'_i \not<: t_i}{\mathbf{recons}\ k(\underline{v})\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ error}}$$

$$\textsc{RC-List-Sucs}\ \frac{\underline{v' \neq \blacksquare}}{\mathbf{recons}\ [\underline{v}]\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ success}\ [\underline{v'}]}$$

$$\textsc{RC-List-Err}\ \frac{v'_i = \blacksquare}{\mathbf{recons}\ [\underline{v}]\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ error}}$$

$$\textsc{RC-Set-Sucs}\ \frac{\underline{v' = \blacksquare}}{\mathbf{recons}\ \{\underline{v}\}\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ success}\ \{\underline{v'}\}}$$

$$\textsc{RC-Set-Err}\ \frac{v'_i = \blacksquare}{\mathbf{recons}\ \{\underline{v}\}\ \mathbf{using}\ \underline{v'}\ \mathbf{to\ error}}$$

$$\textsc{RC-Map-Sucs}\ \frac{\underline{v'' \neq \blacksquare} \qquad \underline{v''' \neq \blacksquare}}{\mathbf{recons}\ (\underline{v : v'})\ \mathbf{using}\ \underline{v''}, \underline{v'''}\ \mathbf{to\ success}\ (\underline{v'' : v'''})}$$

$$\textsc{RC-Map-Err}\ \frac{v''_i = \blacksquare \vee v'''_i = \blacksquare}{\mathbf{recons}\ (\underline{v : v'})\ \mathbf{using}\ \underline{v''}, \underline{v'''}\ \mathbf{to\ error}}$$

$$\textsc{RC-Bot-Sucs}\ \frac{}{\mathbf{recons}\ \blacksquare\ \mathbf{using}\ \varepsilon\ \mathbf{to\ success}\ \blacksquare}$$

$$\textsc{RC-Bot-Err}\ \frac{}{\mathbf{recons}\ \blacksquare\ \mathbf{using}\ v', \underline{v''}\ \mathbf{to\ error}}$$

Enumeration The enumeration relation $\boxed{e; \underline{\rho}; \sigma \underset{\text{each}}{\Longrightarrow} vres; \sigma'}$ iterates over all provided bindings (EE-More-Sucs) until there are none left (EE-Emp) or the result is neither an ordinary value or **continue** from one of the iterations (EE-More-Exc); in case the result is **break** the evaluation will terminate early with a succesful result (EE-More-Break).

$$\text{EE-Emp}\ \frac{}{e; \varepsilon; \sigma \underset{\text{each}}{\Longrightarrow} \textbf{success}\ \blacksquare; \sigma}$$

$$\text{EE-More-Sucs}\ \frac{\begin{array}{c} e; \sigma\rho \underset{\text{expr}}{\Longrightarrow} vres; \sigma'' \qquad vres = \textbf{success}\ v \lor vres = \textbf{continue} \\ e; \underline{\rho'}; (\sigma'' \setminus \text{dom}\ \rho) \underset{\text{each}}{\Longrightarrow} vres'; \sigma' \end{array}}{e; \rho, \underline{\rho'}; \sigma \underset{\text{each}}{\Longrightarrow} vres'; \sigma'}$$

$$\text{EE-More-Break}\ \frac{e; \sigma\rho \underset{\text{expr}}{\Longrightarrow} \textbf{break}; \sigma'}{e; \rho, \underline{\rho'}; \sigma \underset{\text{each}}{\Longrightarrow} \textbf{success}\ \blacksquare; (\sigma' \setminus \text{dom}\ \rho)}$$

$$\text{EE-More-Exc}\ \frac{e; \sigma\rho \underset{\text{expr}}{\Longrightarrow} exres; \sigma' \qquad exres \in \{\textbf{throw}\ v, \textbf{return}\ v, \textbf{fail}, \textbf{error}\}}{e; \rho, \underline{\rho'}; \sigma \underset{\text{each}}{\Longrightarrow} exres; (\sigma' \setminus \text{dom}\ \rho)}$$

Generator expressions The evaluation relation for generator expressions has form $\boxed{g; \sigma \underset{\text{gexpr}}{\Longrightarrow} envres; \sigma'}$. For matching assignments the target right-hand side expression is evaluated first—propagating possible exceptional results (G-Pat-Exc) and then the value is matched against target pattern (G-Pat-Sucs).

$$envres ::= \textbf{success}\ \vec{\rho} \mid exres$$

$$\text{G-Pat-Sucs}\ \frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v; \sigma' \qquad \sigma' \vdash p \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \vec{\rho}}{p := e; \sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{success}\ \vec{\rho}; \sigma'}$$

$$\text{G-Pat-Exc}\ \frac{e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}{p := e; \sigma \underset{\text{expr}}{\Longrightarrow} exres; \sigma'}$$

For enumerating assignments, each possible value in a collection is provided as a possible binding to the range variable in the output (G-Enum-List, G-Enum-Set); for maps in particular, only the keys are bounds

(G-Enum-Map). An error is raised if the result value is not a collection (G-Enum-Err), and exceptions are propagated as always (G-Enum-Exc).

$$\textsc{G-Enum-List}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\;[v_1,\ldots,v_\text{n}];\sigma'}{x \leftarrow e;\sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{success}\;[x \mapsto v_1],\ldots,[x \mapsto v_\text{n}];\sigma'}$$

$$\textsc{G-Enum-Set}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\;\{v_1,\ldots,v_\text{n}\};\sigma'}{x \leftarrow e;\sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{success}\;[x \mapsto v_1],\ldots,[x \mapsto v_\text{n}];\sigma'}$$

$$\textsc{G-Enum-Map}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\;(v_1 : v_1',\ldots,v_\text{n} : v_\text{n}');\sigma'}{x \leftarrow e;\sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{success}\;[x \mapsto v_1],\ldots,[x \mapsto v_\text{n}];\sigma'}$$

$$\textsc{G-Enum-Err}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\;v;\sigma' \qquad v = vb \lor v = k(\underline{v'}) \lor v = \blacksquare}{x \leftarrow e;\sigma \underset{\text{gexpr}}{\Longrightarrow} \textbf{error};\sigma'}$$

$$\textsc{G-Enum-Exc}\;\frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} exres;\sigma'}{x \leftarrow e;\sigma \underset{\text{gexpr}}{\Longrightarrow} exres;\sigma'}$$

Pattern Matching

The pattern matching relation $\boxed{v \vdash \sigma \overset{?}{:=} p \underset{\text{match}}{\Longrightarrow} \underline{\rho}}$ takes as input the current store σ, a pattern p and a target value v and produces a sequence of compatible environments that represent possible bindings of variables to values. Pattern matching a basic value against target value produces a single environment that does not bind any variable ([]) if the target value is the same basic value (P-Val-Sucs) and otherwise does not produce any binding environment (ε), (P-Val-Fail).

$$\textsc{P-Val-Sucs}\;\frac{}{\sigma \vdash vb \overset{?}{:=} vb \underset{\text{match}}{\Longrightarrow} []} \qquad \textsc{P-Val-Fail}\;\frac{v \neq vb}{\sigma \vdash vb \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \varepsilon}$$

Pattern matching against a variable depends on whether the variable already exists in the current store. If it is assigned in the current store, then the target value must the assigned value to return a possible binding (P-Var-Uni) and otherwise failing with no bindings (P-Var-Fail); if it is not in the current store, it will simply bind the variable to the target value (P-Var-Bind).

$$\text{P-Var-Uni}\;\frac{x \in \text{dom}\;\sigma \qquad v = \sigma(x)}{\sigma \vdash x \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} []} \qquad \text{P-Var-Fail}\;\frac{x \in \text{dom}\;\sigma \qquad v \neq \sigma(x)}{\sigma \vdash x \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \varepsilon}$$

$$\text{P-Var-Bind}\;\frac{x \notin \text{dom}\;\sigma}{\sigma \vdash x \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} [x \mapsto v]}$$

When pattern matching against a constructor pattern, it is first checked whether the target value has the same constructor. If it does, then the sub-patterns are matched against the contained values of the target value, merging their resulting environments (P-Cons-Sucs), and otherwise failing with no bindings (P-Cons-Fail). The merging procedure is described formally later in this section, but it intuitively it takes the union of all bindings that have consistent assignments to the same variables.

$$\text{P-Cons-Sucs}\;\frac{\sigma \vdash p_1 \overset{?}{:=} v_1 \underset{\text{match}}{\Longrightarrow} \underline{\rho_1} \qquad \ldots \qquad \sigma \vdash p_n \overset{?}{:=} v_n \underset{\text{match}}{\Longrightarrow} \underline{\rho_n}}{\sigma \vdash k(\underline{p}) \overset{?}{:=} k(\underline{v}) \underset{\text{match}}{\Longrightarrow} \text{merge}(\underline{\rho_1}, \ldots, \underline{\rho_n})}$$

$$\text{P-Cons-Fail}\;\frac{v \neq k(\underline{v'})}{\sigma \vdash k(\underline{p}) \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \varepsilon}$$

When pattern matching against a typed labeled pattern, the target value is checked to have a compatible type—failing with no bindings otherwise (P-Type-Fail)— and then then the inner pattern is matched against the same value. The result of the sub-pattern match is merged with the environment where the target value is bound to the label variable (P-Type-Sucs).

$$\text{P-Type-Sucs}\;\frac{v : t' \qquad t' <: t \qquad \sigma \vdash p \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \underline{\rho}}{\sigma \vdash t\; x : p \overset{?}{:=} v \underset{\text{match}}{\Longrightarrow} \text{merge}([x \mapsto v], \underline{\rho})}$$

$$\textsc{P-Type-Fail}\,\frac{v : t' \qquad t' \nless: t}{\sigma \vdash t\ x : p \overset{?}{:=} v \xRightarrow[\text{match}]{} \varepsilon}$$

Pattern matching against a list pattern first checks whether the target value is a list—otherwise failing (P-List-Fail)—and then pattern matches against the sub-patterns returning their result (P-List-Sucs).

$$\textsc{P-List-Sucs}\,\frac{\sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v} \mid \varnothing \xRightarrow[\text{match}\star]{[]\quad ,} \underline{\rho}}{\sigma \vdash [\underline{\star p}] \overset{?}{:=} [\underline{v}] \xRightarrow[\text{match}]{} \underline{\rho}} \qquad \textsc{P-List-Fail}\,\frac{v \neq [\underline{v'}]}{\sigma \vdash [\underline{\star p}] \overset{?}{:=} v \xRightarrow[\text{match}]{} \varepsilon}$$

Pattern matching against a set pattern is analogous to pattern matching against list patterns (P-Set-Sucs, P-Set-Fail).

$$\textsc{P-Set-Sucs}\,\frac{\sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v} \mid \varnothing \xRightarrow[\text{match}\star]{\{\}\quad \uplus} \underline{\rho}}{\sigma \vdash \{\underline{\star p}\} \overset{?}{:=} \{\underline{v}\} \xRightarrow[\text{match}]{} \underline{\rho}} \qquad \textsc{P-Set-Fail}\,\frac{v \neq \{\underline{v'}\}}{\sigma \vdash \{\underline{p}\} \overset{?}{:=} v \xRightarrow[\text{match}]{} \varepsilon}$$

Negation pattern $!p$ matching succeeds with no variables bound if the sub-pattern p produces no binding environment (P-Neg-Sucs), and otherwise fails (P-Neg-Fail).

$$\textsc{P-Neg-Sucs}\,\frac{\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \varepsilon}{\sigma \vdash !p \overset{?}{:=} v \xRightarrow[\text{match}]{} []} \qquad \textsc{P-Neg-Fail}\,\frac{\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \rho \qquad \rho \neq \varepsilon}{\sigma \vdash !p \overset{?}{:=} v \xRightarrow[\text{match}]{} \varepsilon}$$

Descendant pattern matching applies the sub-pattern against target value, and keeps applying the deep matching pattern against the children values, concatenating their results (P-Deep). Notice here the similarities with the deep type-directed match operation in TRON (see Chapter 5); the operation in Rascal Light is however more general allowing use of the complete pattern matching capabilities available, in contrast to only types.

$$\textsc{P-Deep}\,\frac{\begin{array}{c}\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho} \qquad v'_1, \ldots, v'_n = \text{children}(v) \\ \sigma \vdash /p \overset{?}{:=} v'_1 \xRightarrow[\text{match}]{} \underline{\rho'_1} \quad \ldots \quad \sigma \vdash /p \overset{?}{:=} v'_n \xRightarrow[\text{match}]{} \underline{\rho'_n}\end{array}}{\sigma \vdash /p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho}, \underline{\rho'_1}, \ldots, \underline{\rho'_n}}$$

The star pattern matching relation has form $\boxed{\sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho}}$ which tries to match the sequence of patterns $\underline{\star p}$ on the left-hand side with sequence of provided values $\underline{v}$ on the right-hand side; the relations is parameterized over the construction function ($\langle\rangle$) and the partition relation ($\otimes$), and because matching arbitrary elements patterns $\star x$ is non-deterministic we keep track of a set of values $\mathbb{V}$ that have already been tried for the latest available variable.

If both the pattern sequence and the value sequence is empty, then we have successfully finished matching (PL-Emp-Both). Otherwise if any of the sequences finish while the other is non-empty we produce no possible bindings (PL-Emp-Pat, PL-Emp-Val); an exemption is made for arbitrary match patterns because they can match empty sequences of values.

$$\text{PL-Emp-Both}\ \frac{}{\sigma \vdash \varepsilon \overset{?}{:=} \varepsilon \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} [\,]} \qquad \text{PL-Emp-Pat}\ \frac{}{\sigma \vdash \varepsilon \overset{?}{:=} v, \underline{v'} \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \varepsilon}$$

$$\text{PL-Emp-Val}\ \frac{}{\sigma \vdash p, \underline{\star p} \overset{?}{:=} \varepsilon \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \varepsilon}$$

When the initial element of the starred pattern sequence is an ordinary pattern then it is matched against the initial element of the value sequence, and the rest of the pattern sequence is matched against the rest of the value sequence. The results of both submatches are then merged together (PL-More-Pat).

$$\text{PL-More-Pat}\ \frac{\sigma \vdash p \overset{?}{:=} v \xrightarrow[\text{match}]{} \underline{\rho} \qquad \sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v'} \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho'}}{\sigma \vdash p, \underline{\star p} \overset{?}{:=} v, \underline{v'} \mid \mathbb{V} \xrightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \text{merge}(\underline{\rho}, \underline{\rho'})}$$

Like ordinary variables, arbitrary matching patterns depend on whether the binding variable already exists in the current store. If the variable is assigned in the current store then either there must exist a partition of values that has a matching subcollection (PL-More-Star-Uni), or the matching fails producing any consistent binding environment (PL-More-Star-Pat-Fail, PL-More-Star-Val-Fail).

$$\textsc{PL-More-Star-Uni}\ \dfrac{\begin{array}{c} x \in \operatorname{dom} \sigma \quad \sigma(x) = \langle \underline{v'} \rangle \quad \underline{v} = \underline{v'} \otimes \underline{v''} \\ \sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v''} \mid \varnothing \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho} \end{array}}{\sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho}}$$

$$\textsc{PL-More-Star-Pat-Fail}\ \dfrac{x \in \operatorname{dom} \sigma \quad \sigma(x) = \langle \underline{v'} \rangle \quad \nexists \underline{v''}. \underline{v} = \underline{v'} \otimes \underline{v''}}{\sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \varepsilon}$$

$$\textsc{PL-More-Star-Val-Fail}\ \dfrac{x \in \operatorname{dom} \sigma \quad \sigma(x) = v \quad v \neq \langle \underline{v'} \rangle}{\sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v''} \mid \mathbb{V} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \varepsilon}$$

If the variable is not in the current store then there are two options: either i) there still exist a partition that is possible to try, or ii) we have exhausted all possible partitions of the value sequence. In the first case, an arbitrary partition is bound to the target variable and the rest of the patterns are matched against the rest of the values, merging their results; additionally, the other partitions are also tried concatenating their results with the merged one (PL-More-Star-Re). In the exhausted case, the pattern match produces no bindings (PL-More-Star-Exh).

$$\textsc{PL-More-Star-Re}\ \dfrac{\begin{array}{c} x \notin \operatorname{dom} \sigma \quad \underline{v} = \underline{v'} \otimes \underline{v''} \quad \underline{v'} \notin \mathbb{V} \\ \sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v''} \mid \varnothing \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho} \\ \sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \cup \underline{v'} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \underline{\rho'} \end{array}}{\sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \operatorname{merge}([x \mapsto \underline{v'}], \underline{\rho}), \underline{\rho'}}$$

$$\textsc{PL-More-Star-Exh}\ \dfrac{x \notin \operatorname{dom} \sigma \quad \nexists \underline{v'}, \underline{v''}. \underline{v} = \underline{v'} \otimes \underline{v''} \wedge \underline{v'} \notin \mathbb{V}}{\sigma \vdash \star x, \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xRightarrow[\text{match}\star]{\langle\rangle \quad \otimes} \varepsilon}$$

Merging a sequence of possible variable bindings produces a sequence containing consistent variable bindings from the sequence: that is, all possible environments that assign consistent values to the same variables are merged.

$$\begin{aligned}\text{merge}(\varepsilon) &= [] \\ \text{merge}(\underline{\rho}, \underline{\rho'_1}, \ldots, \underline{\rho'_n}) &= \text{merge-pairs}(\underline{\rho} \times \text{merge}(\underline{\rho'_1}, \ldots, \underline{\rho'_n}))\end{aligned}$$

$$\text{merge-pairs}(\langle \rho_1, \rho'_1 \rangle, \ldots, \langle \rho_n, \rho'_n \rangle) = \text{merge-pair}(\rho_1, \rho'_1), \ldots, \text{merge-pair}(\rho_n, \rho'_n)$$

$$\text{merge-pair}(\rho, \rho') = \begin{cases} \rho\rho' & \textbf{if } \forall x \in \text{dom } \rho \cap \text{dom } \rho'.\rho(x) = \rho'(x) \\ \varepsilon & \textbf{otherwise} \end{cases}$$

6.3 Semantics Properties

Backtracking is pure in Rascal (Light) programs, and so if evaluating a set cases produces **fail** as result, the initial state is restored.

Theorem 6.1 (Backtracking purity). *If* $\overset{\mathcal{CS}}{\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} \textbf{fail}; \sigma'}$ *then* $\sigma' = \sigma$

Strong typing is an important safety property of Rascal, which I capture by specifying a theorems with two result properties: one about the well-typedness of state, and the other about well-typedness of resulting values.

Theorem 6.2 (Strong typing). *Assume that semantic unary* $[\![\ominus]\!]$ *and binary operators* $[\![\oplus]\!]$ *are strongly typed. If* $\overset{\mathcal{E}}{e; \sigma \xRightarrow[\text{expr}]{} vres; \sigma'}$ *and there exists a type* t *such that* $\overset{\mathcal{T}}{v : t}$ *for each value in the input store* $v \in \text{img } \sigma$*, then*

1. *There exists a type* t' *such that* $v' : t'$ *for each value in the result store* $v' \in \text{img } \sigma'$*.*

2. *If the result value* $vres$ *is either* **success** v''*,* **return** v''*, or* **throw** v''*, then there exists a type* t'' *such that* $v'' : t''$*.*

Consider an augmented version of the operational semantics where each execution relation is annotated with *partiality fuel* [Amin and Rompf, 2017]—represented by a superscript natural number n—which specifies the maximal number of recursive premises allowed in the derivation. The fuel is subtracted on each recursion, resulting in a **timeout** value when it reaches zero, and the rule set is amended by congruence rules which propagate **timeout** results from the recursive premises to the conclusion.

$$\mathit{vtres} ::= \mathit{vres} \mid \textbf{timeout}$$

For this version of the semantics, we can specify the property that execution will either produce a result or it will timeout; that is, the semantics does not get stuck.

Theorem 6.3 (Partial progress)**.** *It is possible to construct a derivation* $e; \sigma \underset{\text{expr}}{\Longrightarrow}^{n} \mathit{vtres}; \sigma'$ *for any input expression* e*, well-typed store* σ *and fuel* n*.*

Finally, consider a subset of the expression language described by syntactic category e_{fin}, where **while**-loops, **solve**-loops and function calls are disallowed, and similarly with all traversal strategies except **bottom-up** and **bottom-up-break**. This subset is known to be terminating:

Theorem 6.4 (Terminating expressions)**.** *There exists* n *such that derivation* $\overset{\mathcal{E}}{e_{\text{fin}}; \sigma \underset{\text{expr}}{\Longrightarrow}^{n} \mathit{vres}; \sigma'}$ *has a result* vres *which is not* **timeout** *for expression* e_{fin} *in the terminating subset.*

Remark 6.1. Why is the **top-down** traversal strategy potentially non-terminating while the **bottom-up** traversal strategy is terminating? The answer lies in that the **top-down** traverses the children of the target expression after evaluating the required case, which makes it possible to keep add children ad infinitum as illustrated by the following example:

```
data Nat = zero() | succ(Nat pred);

Nat infincrement(Nat n) =
  top-down visit(n) {
     case succ(n) => succ(succ(n))
  }
```

6.4 Formal Semantics, Types and Intented Semantics

This formal specification of Rascal Light was largely performed when the official type checker of Rascal (developed by CWI Amsterdam) was only at the experimental stage. Useful extension of this formalization include a deductive type system that includes polymorphic aspects of Rascal to prove the consistency of the type system, and type safety with regards to the dynamic semantics provided in this formalization.

The presented semantics was checked against the Rascal implementation in various ways: the existing implementation was used as a reference point, rules for individual were held up against the documentation, and correspondence with the Rascal developers was used to clarify remaining questions. A more formal way to check whether the captured semantics is the intended one, is to construct an independent formal semantics which is related to the natural semantics presented in this chapter. This could be an axiomatic semantics Hoare [1969], which modularly specifies for each construct the intented effects using logical formulae as pre and post-conditions, without necessarily specifying how each construct is evaluated. The natural semantics would then be checked to satisfy the axiomatic semantics for each construct, which will further increase confidence that the captured semantics is the intented one.

6.5 Recap

I presented the formalization of a large subset of the operational part of Rascal Klint et al. [2009, 2011], called Rascal Light. The formalization was primarily based on the available open source implementation[4] and the accompanying documentation[5], and personal correspondence with the developers further clarified previous ambiguities and mismatches.

Unlike TRON, Rascal Light is meant to be both a formal and practical language subset, and so supports a much broader range of both ordinary language features—e.g., function definitions (including recursion) loops (including control flow operators), case analysis, and exceptions—and high-level transformation language features such as generic traversals with strategies, backtracking and powerful pattern matching. This subset has been verified against properties specifying strong typing and safety, which provides further trustworthiness.

[4]https://github.com/usethesource/rascal
[5]http://tutor.rascal-mpl.org/

Chapter 7

Verifying Rascal Transformations using Inductive Refinement Types

The established rich features in Rascal—first-class traversal, powerful pattern matching, backtracking, fixed-point iteration— make it easy to express complex programs in a succinct and maintainable manner. Verification-wise, availability of these features is a double-edged. They provide an opportunity to develop techniques that provide more efficient and precise results for the different constructs, e.g., by better propagating information during traversal and provide specialized abstractions for the different collections available. They however also bring along challenges:

1. The manipulated data represents often large models or programs, relying on large inductive algebraic data types and collections such as lists, sets and maps.

2. There is a non-trivial control flow stemming from the presence of exceptions and backtracking—allowing potentially non-local jumps during execution—and the traversal constructs, where the control flow depends on the type and shape of mached data in addition to the current state.

In this chapter, we will discuss how we combine existing techniques in abstract interpretation in order to deal with the aforementioned challenges. Concretely, we will discuss the following contributions:

- Modular domain design for abstract shape domains, that allows easily extending and replacing abstraction for concrete types of elements, e.g. extending the abstraction for lists to include length of elements in addition to shapes.

- Formal Schmidt-style abstract *operational* semantics [Schmidt, 1998] for a subset of Rascal, tuning the idea of trace-based memoization to support the various expressive constructs in Rascal.
- An abstract interpretation-based static analysis tool for Rascal Light that supports inferring inductive refinements types that characterise the type and shape of manipulated data.
- An evaluation of the technique on a series of realistic program transformations, including a desugaring transformation and a DSL transpilation transformation taken from real Rascal projects.

These contributions together show the feasibility of using abstract interpretation for constructing static analyses for expressive transformation languages and properties. We hope that our presented adaptation of Schmidt-style abstract interpretation provides inspiration beyond Rascal, and that more language designers will reuse their existing interpreters of other languages to build similar tools.

7.1 Motivating Example

As a motivating example, we will use a multiplicative expression simplification transformation (Figure 7.1) which eliminates multiplications by 0 and 1. The example is inspired by the constant folding transformation common in many compilers, and is implemented in Rascal using bottom-up traversal for the required rewriting of sub-expressions.

```
data Nat = zero() | suc(Nat pred);
data Expr = var(str nm) | cst(Nat vl) | mult(Expr el, Expr er);

Expr simplify(Expr expr) =
  bottom-up visit (expr) {
      case mult(cst(zero()), y) => cst(zero())
      case mult(x, cst(zero())) => cst(zero())
      case mult(cst(suc(zero())), y) => y
      case mult(x, cst(suc(zero()))) => x
  };
```

Figure 7.1: Expression Simplification

For similar transformations to this one, there are typically multiple properties we want to verify:

Type properties concern that all executions of the program are type and scope-safe. In our example, this would be that the result of the visitor is of type `Expr` as expected by the `simplify` function.

Shape properties concern that the output data of our program has a particular structure. For our example, this could be no multiplication sub-expression `mult` in its output which has `cst(zero())` or `cst(suc(zero()))` as a sub-expression.

The presented technique allows verifying such type and shape properties by inferring inductive refinements of algebraic data type definitions, e.g., the Expr data type used in the motivating example. To illustrate this, we will abstractly interpret our motivating example, providing a high-level overview of the required steps:

1. Let the initial abstract store be $\widehat{\sigma} = [\mathrm{expr} \mapsto \mathrm{Expr}]$, so the input variable expr abstractly represents any value of type Expr.
2. Execution proceeds by evaluating the **visit**-statement on Line 5; since the traversal is bottom-up, it starts traversing the contained values (children) and then the applies the required cases to the top-level value.
   a) The children of a datatype value dependent on the particular constructor used, and so the interpreter proceed to unfold expr to the possible constructor values: $\mathrm{mult}(\mathrm{Expr}, \mathrm{Expr})$, $\mathrm{cst}(\mathrm{Nat})$ and $\mathrm{var}(\mathbf{str})$.
   b) The interpreter considers each of the possible constructor values separately, merging their results using the least upper bound operator when execution is finished.
3. In case the analyzed value is $\mathrm{mult}(\mathrm{Expr}, \mathrm{Expr})$ then the children have type Expr twice.
   a) Traversing the first child requires us to recursively consider the type Expr again, which would result in non-termination if the above steps were naively repeated.

b) Instead we will apply a version of trace memoization [Schmidt, 1998] where we return the value of the previous recursive traversal to compute a new approximation of the resul, and then recompute the recursive traversal again until a fixed-point is reached.

c) Because this is the first recursion in the traversal, we will return the bottom value $\perp$ is used as state and the traversal continues evaluation at the other branches.

4. In case the analyzed value is cst(Nat) then there is a single child of type Nat.

   - Traversal proceeds with the type Nat, using a similar procedure of unfolding, merging, memoization and fixed-point iteration as above.
   - Since there is no case that affects the type Nat, the traversal produces **fail** Nat; $\widehat{\sigma}$ as a result, signifying that the evaluation did not match any case. Notice, that the failure result of case evaluation, additionally contains an abstract value Nat—not present in the concrete executor—which signifies a refinement of the input value.
   - We have to use the result to reconstruct an updated value using the cst constructor for the rest of the traversal, which in this case trivially succeeds producing the same value as our input cst(Nat).
   - Since cst(Nat) does not match any case either, we get **fail** cst(Nat); $\widehat{\sigma}$ as a result for this branch.

5. In case of var(**str**) the traversal proceeds in the same way as the cst case, with none of the cases matching and a result value of **fail** var(**str**); $\widehat{\sigma}$.

6. We merge the result of the three branches which produces $\perp \sqcup (\textbf{fail}\ \mathrm{cst}(\mathrm{Nat}); \widehat{\sigma}) \sqcup (\textbf{fail}\ \mathrm{var}(\textbf{str}); \widehat{\sigma}) = \textbf{fail}\ \mathrm{Expr\#1}; \widehat{\sigma}$ as result, where:

$$\textbf{refine}\ \mathrm{Expr\#1} = \mathrm{cst}(\mathrm{Nat}) \mid \mathrm{var}(\textbf{str})$$

   This refinement of input values on failure is important to maintain precision during traversals, which is what allows us to produce precise result shapes. In particular, if we did not produce a refinement as above, we would only have the input value shape (Expr) and so we would not be able to infer any useful shape property.

7. Since we got a new result that was more over-approximating than the previous state—i.e., $\perp \sqsubseteq (\textbf{fail}\ \text{Expr\#1}; \widehat{\sigma})$—the trace memoization technique requires us to re-execute the **visit**-statement, in order to maintain soundness (via over-approximation).

8. We unfold to three possible constructor cases again: mult(Expr, Expr), cst(Nat) and var(**str**). We only have to reconsider the result for mult(Expr, Expr) since the result for the other two cases remain the same.

9. In the case mult(Expr, Expr), so the children are of type Expr, twice.

   a) This time on the recursive traversal of Expr, trace memoization will return the state from the previous iteration **fail** Expr#1;$\widehat{\sigma}$.
   b) We reconstruct the expression with the refined children values, to get value mult(Expr#1, Expr#1)
   c) We unfold and match the cases, applying the right-hand sides and get the following results:
      i. On the first two cases (Line 6 and Line 7), we get **success** cst(zero());$\widehat{\sigma}$ as a result.
      ii. On the second two cases (Line 8 and Line 9), we get **success** Expr#2;$\widehat{\sigma}$ as a result, where:

$$\textbf{refine}\ \text{Expr\#2} = \text{cst}(\text{suc}(\text{Nat})) \mid \text{var}(\textbf{str})$$

      since both the left and right component are refined to not be cst(zero()) after matching the previous patterns, and the multiplication with cst(suc(zero())) is rewritten to only contain the other operand.
      iii. In case of failure, we get **fail** mult(Expr#3, Expr#3);$\widehat{\sigma}$ as a result, where:

$$\textbf{refine}\ \text{Expr\#3} = \text{cst}(\text{suc}(\text{suc}(\text{Nat}))) \mid \text{var}(\textbf{str})$$

      since both components are also refined to not be suc(zero()) after matching the previous patterns.

d) Combining the results for the different cases by taking the least upper bound gives us

$$(\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{mult}(\mathrm{Expr\#3}, \mathrm{Expr\#3}); \widehat{\sigma})$$

as a result state which keeps track of separate shapes and stores for each type of result (success and failure).

10. Then by combining all constructor branch results we get:

$$(\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{Expr\#4}; \widehat{\sigma})$$

where we have

$$\textbf{refine}\ \mathrm{Expr\#4} = \mathrm{cst}(\mathrm{Nat}) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#3}, \mathrm{Expr\#3})$$

11. Since we got a new more over-approximating result from previous execution of **visit**-statement—i.e., $(\textbf{fail}\ \mathrm{Expr\#1}) \sqsubseteq (\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{Expr\#4}; \widehat{\sigma})$—the fixed-point iteration must be continued. To ensure termination in a finite number of steps, we use widening between the previous result state and the current result state $(\textbf{fail}\ \mathrm{Expr\#1}; \widehat{\sigma}) \nabla (\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{Expr\#4}; \widehat{\sigma}) = (\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{Expr\#4}; \widehat{\sigma})$, which produces the same state as the new result because the previous result did not have any recursively defined refinement.

12. After another iteration for the fixed-point computation of **visit**-statement we get $(\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{fail}\ \mathrm{Expr\#5}; \widehat{\sigma})$ as result, where:

$$\textbf{refine}\ \mathrm{Expr\#5} = \mathrm{cst}(\mathrm{Nat}) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#6}, \mathrm{Expr\#6})$$
$$\textbf{refine}\ \mathrm{Expr\#6} = \mathrm{cst}(\mathrm{suc}(\mathrm{suc}(\mathrm{Nat}))) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#3}, \mathrm{Expr\#3})$$

13. By widening of the new result with the previous one, we get a result state $(\textbf{success}\ \mathrm{Expr\#1}; \widehat{\sigma}), (\textbf{success}\ \mathrm{Expr\#6}; \widehat{\sigma})$. This uses the widening algorithm presented in Section 7.3, to calculate the widening on refinements: $\mathrm{Expr\#4} \nabla \mathrm{Expr\#5} = \mathrm{Expr\#7}$ with previous inductive refinement definitions:

$$\textbf{refine}\ \mathrm{Expr\#3} = \mathrm{cst}(\mathrm{suc}(\mathrm{suc}(\mathrm{Nat}))) \mid \mathrm{var}(\textbf{str})$$
$$\textbf{refine}\ \mathrm{Expr\#4} = \mathrm{cst}(\mathrm{Nat}) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#3}, \mathrm{Expr\#3})$$
$$\textbf{refine}\ \mathrm{Expr\#5} = \mathrm{cst}(\mathrm{Nat}) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#6}, \mathrm{Expr\#6})$$
$$\textbf{refine}\ \mathrm{Expr\#6} = \mathrm{cst}(\mathrm{suc}(\mathrm{suc}(\mathrm{Nat}))) \mid \mathrm{var}(\textbf{str}) \mid \mathrm{mult}(\mathrm{Expr\#3}, \mathrm{Expr\#3})$$

and new refinement definitions:

$$\begin{aligned} \textbf{refine}\ \text{Expr\#7} &= \text{cst}(\text{Nat}) \mid \text{var}(\textbf{str}) \mid \text{mult}(\text{Expr\#8}, \text{Expr\#8}) \\ \textbf{refine}\ \text{Expr\#8} &= \text{cst}(\text{suc}(\text{suc}(\text{Nat}))) \mid \text{var}(\textbf{str}) \mid \text{mult}(\text{Expr\#8}, \text{Expr\#8}) \end{aligned}$$

The core intuition behind the refinement is to combine the set of constructors, recursively merging the arguments of the common ones; additionally, when merging two inductive refinements, we create a new refinement replacing the references to the old ones with it. In our example, this means that merging Expr#3 and Expr#6 creates Expr#8 which has the same set of constructors, but the recursive references are to Expr#8 instead of Expr#3 or Expr#6.

14. After another iteration, we get the same result $(\textbf{success}\ \widehat{\text{Expr\#1}}), (\textbf{fail}\ \text{Expr\#7}; \widehat{\sigma})$ and so we have reached a fixed-point.

15. On failure, the **visit**-expression after traversal acts as a kind of identity transformation producing a successful rules containing the original input value. In our abstract interpreter we will do the same on failure, but use the refined the result; we therefore convert the possible failure state **fail** Expr#7; $\widehat{\sigma}$ to **success** Expr#7; $\widehat{\sigma}$ and merge with the other success state **success** Expr#1; $\widehat{\sigma}$ to get **success** Expr#7; $\widehat{\sigma}$ as final result for the **visit**-expression.

On the result inductive shape, we can then perform checks whether our target properties are satisfied. For example, we can easily see that all constants have natural numbers of shape suc(suc(Nat)) (that is, ≥ 2), and so the simplification procedure correctly eliminates all multiplications by 0 and 1.

7.2 Formal Language

The presented technique is meant to work with all of Rascal Light, but to keep the formal presentation concise we will consider a further subset, called Rascal UltraLight. We will deliminate this subset in this section, and then use it to describe the general technique we use to derive abstract operational semantics from the concrete operational semantics for

Rascal Light.[1] The constructs in this subset get their formal semantics directly from the Rascal Light concrete semantics presented in Chapter 6, but we will recall the rules when appropriate.

We consider the following value types: integers (**int**), finite sets ($\mathbf{set}\langle t\rangle$) and algebraic data types (at). Each algebraic data type, at associates a set of unique constructors, so that each constructor $k(\underline{t})$ has a fixed set of typed parameters.

$$vt \in \text{ValueType} ::= \ \mathbf{int} \mid \mathbf{set}\langle t\rangle \mid at \in \text{TyCons}$$

Like the types in Rascal, there is support for subtyping where we have **void** and **value** as bottom and top types specifically.

$$t \in \text{Type} ::= \ \mathbf{void} \mid vt \mid \mathbf{value}$$

The value domain Value is generated by the least fixedpoint of the following semantic equations for the types:

$$[\![\mathbf{void}]\!] = \emptyset \quad [\![\mathbf{int}]\!] = \mathbb{Z} \quad [\![\mathbf{set}\langle t\rangle]\!] = \wp([\![t]\!]) \quad [\![\mathbf{value}]\!] = \bigcup_{vt}[\![vt]\!]$$
$$[\![at]\!] = \left\{k(\underline{v}) \;\middle|\; k(\underline{t}) \text{ is a constructor of } at \wedge \forall i.v_i \in [\![t_i]\!]\right\}$$

Expression language We consider the following subset of Rascal expressions:

$$\begin{gathered} e ::= n \in \mathbb{Z} \mid x \in \text{Var} \mid e \otimes e \mid x = e \mid e;e \mid k(\underline{e}) \mid \{\underline{e}\} \\ \mid \mathbf{fail} \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{bottom\text{-}up\ visit}\ e\ \underline{cs} \\ cs ::= \mathbf{case}\ p \Rightarrow e \qquad p ::= x \mid k(\underline{p}) \mid \{\underline{\star p}\} \qquad \star p ::= p \mid \star x \end{gathered}$$

The considered subset includes integer constants, variables, binary operators, variable assignments, sequencing, constructor expressions, set literal expressions, matching failure, conditionals and bottom-up visitors. The pattern language consists of either variable patterns x, constructor patterns $k(\underline{p})$ or set patterns $\{\underline{\star p}\}$ which internally consists of either ordinary patterns or star patterns $\star x$, which match a subcollection and binds the value to the variable x.

[1]We will even for the Rascal UltraLight subset focus on presenting the high-level ideas behind the general technique and not mechanically translate every rule and auxiliary function.

7.3 Abstract Domains

Our abstract domain design focuses on providing a modular way to compose individual abstract domains, such that it supports the following properties:

Replacement Allowing changes to the abstract domain for a particular concrete type of elements without affecting the rest of the complete abstract domain, e.g. replacing the interval abstract domain with the congruence abstract domain for integers.

Extension Adding abstract domains for new types of values, e.g. supporting an abstraction for lists of elements.

Operational Compositionality Domain operations (e.g., least upper bound, subsumption checking and widening) can be performed effectively in a compositional fashion, e.g., that the widening for lists does not dependent on the definition of the particular widening used for integers.

Our goal is to construct an abstract value domain $\widehat{v} \in \widehat{\text{Value}}$ which captures key shape and type properties for the different values in our concrete domain (integers, set, algebraic data types). As with our motivating example, this includes capturing inductive shapes represented by inductive data type refinements of form:

$$\textbf{refine}\ at\#r = k_1(\underline{\widehat{v}_1}) \mid \cdots \mid k_{\mathrm{n}}(\underline{\widehat{v}_{\mathrm{n}}})$$

where the left hand side defines a refinement with name *at#r* that refines a datatype *at* with a subset of possible constructors on the right handside that have abstract value components refining the shape of elements that are abstracted and can possibly inductively refer to *at#r* and other refinements.

The modular abstract domain design generalizes parameterized abstract domains initially suggested by Cousot [2003] to follow a design inspired by the modular construction of types and domains suggested in various literature in type theory and domain theory [Scott, 1976; Smyth and Plotkin, 1982; Winskel, 1993b; Löh and Magalhães, 2011; Chapman et al., 2010; Backhouse et al., 2007; Benke et al., 2003]. The idea is to define domains parametrically—i.e. of form $\widehat{\mathrm{F}}(\underline{\widehat{\mathrm{E}}})$ so that abstract domains for subcomponents are taken as parameters, and instead of having closed recursion the recursive references are replaced by an abstract domain parameter which is decided later. After defining the abstract domains parametrically for each concrete domain we wish to abstract, we can then use standard domain combinators [Winskel, 1993b] —products, sums, fixed points—to combine the various domains into our target abstract value domain. We assume it is possible to pass information for operations recursively as long as it is done in a parametric fashion, so the information only affects operations for elements of the same domain; to avoid an overly abstract presentation we will only mention the relevant information to be passed along when needed.

Integer shape domain

To abstract over integer values, we use the classic interval domain [Cousot and Cousot, 1977]. We chose this domain because it is well-defined, familiar and easy to use with all arithmetic operations; because of our modular design, it is easy to replace it when needed with another suitable numeric abstract domain such as sign, parity or congruence[2].

An interval $[l; u] \in \widehat{\text{Interval}}$ represents a continuous set of integers $\{n \in \mathbb{Z} \mid l \leq n \leq u\}$ and we let $n \in \mathbb{Z}$ refer to the singleteon interval $[n; n]$. The $\widehat{\text{Interval}}$ domain forms an infinite lattice, with the following operations:

$$\begin{gathered}
\bot_{\widehat{\mathrm{I}}} = [\infty; -\infty] \qquad \top_{\widehat{\mathrm{I}}} = [-\infty; \infty] \\
[l_1; u_1] \sqsubseteq_{\widehat{\mathrm{I}}} [l_2; u_2] \ \textbf{iff} \ l_2 \leq l_1 \wedge u_1 \leq u_2 \\
[l_1; u_1] \sqcup_{\widehat{\mathrm{I}}} [l_2; u_2] = [\min(l_1, l_2); \max(u_1, u_2)] \\
[l_1; u_1] \sqcap_{\widehat{\mathrm{I}}} [l_2; u_2] = [\max(l_1, l_2); \min(u_1, u_2)]
\end{gathered}$$

The $\widehat{\text{Interval}}$ also admits a *widening*, which is an upper bound of elements that ensures finite convergence to a result when iteratively applied:

$$[l_1; u_1] \ \nabla_{\widehat{\mathrm{I}}} \ [l_2; u_2] = [\text{if}(l_2 < l_1, -\infty, l_1); \text{if}(u_1 < u_2, \infty, u_1)]$$

[2]In theory one could allow even relational domains such as octagon, but it would require a redesign of the presented modular framework.

Proposition 7.1. *The* $\widehat{\text{Interval}}$ *domain forms a Galois connection with the power set of integers* $\wp(\mathbb{Z}) \underset{\alpha_{\widehat{\mathrm{I}}}}{\overset{\gamma_{\widehat{\mathrm{I}}}}{\leftrightarrows}} \widehat{\text{Interval}}$*, where:*

$$\alpha_{\widehat{\mathrm{I}}}(N) = [min(N); max(N)] \qquad \gamma_{\widehat{\mathrm{I}}}([l;u]) = \{n \in \mathbb{Z} \mid l \leq n \leq u\}$$

Set Shape Domain

Recall that the goal of our analysis is to capture type and shape properties, and so we must pick an abstraction that can reason about the shape of elements inside sets (and other collections) to preserve precision, especially for inductive refinements; additionally, operations on collections like pattern matching and iteration rely on the cardinality of collections and it is likewise useful to provide an abstraction for this component.

We therefore abstract finite sets using a parameterized abstract domain, $\widehat{\text{SetShape}}(\widehat{\mathrm{E}})$, consisting of a reduced product between a component that abstracts the shape of elements $\widehat{\text{SetContent}}(\widehat{\mathrm{E}})$ and a component that abstracts the cardinality of the set using non-negative intervals $\widehat{\text{Interval}}^{+}$. The set shape content abstraction $\widehat{\text{SetContent}}(\widehat{\mathrm{E}})$ is defined as follows:

$$\widehat{sc} \in \widehat{\text{SetContent}}(\widehat{\mathrm{E}}) ::= \{\widehat{e}\},\ \widehat{e} \in \widehat{\mathrm{E}}$$

We will use the notation $\{\widehat{e}\}_{[l;u]}$, where $\widehat{e} \in \widehat{\mathrm{E}}$ describes the content and $[l;u] \in \widehat{\text{Interval}}^{+}$ describes the cardinality to represent $(\{\widehat{e}\}, [l;u]) \in \widehat{\text{SetShape}}(\widehat{\mathrm{E}})$.

Example 7.1. Using the $\widehat{\text{Interval}}$ domain as content abstract domain, we can abstract sets of integers using $\widehat{\text{SetShape}}(\widehat{\text{Interval}})$ as follows:

$$\alpha_{\widehat{\mathrm{SS}}}(\{\{1,42\},32\}) = \{[1;42]\}_{[1;2]} \qquad \alpha_{\widehat{\mathrm{SS}}}(\{\{-41\},\emptyset\}) = \{-41\}_{[0;1]}$$

▲

If the content parameter domain $\widehat{\mathrm{E}}$ forms a lattice (with widening), then we can lift all operations trivially to this domain:

$$\bot_{\widehat{\mathrm{SC}}} = \{\bot_{\widehat{\mathrm{E}}}\} \qquad \top_{\widehat{\mathrm{SC}}} = \{\top_{\widehat{\mathrm{E}}}\} \qquad \{\widehat{e_1}\} \sqsubseteq_{\widehat{\mathrm{SC}}} \{\widehat{e_2}\} \textbf{ iff } \widehat{e_1} \sqsubseteq_{\widehat{\mathrm{E}}} \widehat{e_2}$$
$$\{\widehat{e_1}\} \sqcup_{\widehat{\mathrm{SC}}} \{\widehat{e_2}\} = \{\widehat{e_1} \sqcup_{\widehat{\mathrm{E}}} \widehat{e_2}\} \qquad \{\widehat{e_1}\} \sqcap_{\widehat{\mathrm{SC}}} \{\widehat{e_2}\} = \{\widehat{e_1} \sqcap_{\widehat{\mathrm{E}}} \widehat{e_2}\}$$
$$\{\widehat{e_1}\} \nabla_{\widehat{\mathrm{SC}}} \{\widehat{e_2}\} = \{\widehat{e_1} \nabla_{\widehat{\mathrm{E}}} \widehat{e_2}\}$$

Given a Galois connection for the content domains $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$ we produce the following abstraction $\alpha_{\widehat{\mathrm{SS}}}$ and concretization $\gamma_{\widehat{\mathrm{SS}}}$ between the concrete set of elements domain $\wp(\wp(\mathrm{E}))$ and abstract set shape domain $\widehat{\mathrm{SetShape}}(\widehat{\mathrm{E}})$:

$$\alpha_{\widehat{\mathrm{SS}}}(ess) = \bigsqcup_{es \in ess} \{\alpha_{\widehat{\mathrm{E}}}(es)\}_{|es|}$$

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

Theorem 7.1. *Assuming Galois connection for element domains* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$ *then we have a Galois connection for the set shape domain* $\wp(\wp(\mathrm{E})) \underset{\alpha_{\widehat{\mathrm{SS}}}}{\overset{\gamma_{\widehat{\mathrm{SS}}}}{\leftrightarrows}} \widehat{\mathrm{SetShape}}(\widehat{\mathrm{E}})$

Data Shape Domain

For the clear majority of transformations, we want to ensure that the output captures the desired type and shape properties at all levels of the abstract syntax, i.e., that a program is fully normalized or that all subcomponents of a model have been suitably translated. A particularly natural way to capture this is by representing the shape constraints inductively, which we do using inductive refinements.

Inductive refinements are a generalization of refinement types [Freeman and Pfenning, 1991; Xi and Pfenning, 1998; Rushby et al., 1998] using ideas from regular tree abstractions [Aiken and Murphy, 1991; Cousot and Cousot, 1995]. Like refinement types, we allow constraining the constructors and content—in our case using classical abstract domains like the aforementioned interval—and we build on the regular tree operations to make it possible to infer inductive shapes.

Concretely, our abstraction of data type values is done using the abstract domain $\widehat{\mathrm{DataShape}}(\widehat{\mathrm{E}})$, which takes a parameter domain $\widehat{\mathrm{E}}$ that abstracts the shape of elements in constructors. We define the elements of the data shape domain as follows:

$$\widehat{d} \in \widehat{\mathrm{DataShape}}(\widehat{\mathrm{E}}) ::= \bot_{\widehat{\mathrm{DS}}} \mid at\#r \in \widehat{\mathrm{Refinement}} \mid \top_{\widehat{\mathrm{DS}}}$$

We have the bottom $\bot_{\widehat{\mathrm{DS}}}$ and top $\top_{\widehat{\mathrm{DS}}}$ elements—respectively representing no data types value and all data type values— and references to inductive refinements *at#r* abstracting over the values of algebraic data type *at*. For our parametrized data shape domain, the corresponding definition of refinement *at#r* consists of a choice between a subset of constructors of *at* where the arguments are abstracted using the content domain $\widehat{\mathrm{E}}$:

$$\textbf{refine } at\#r_1 = k_1(\underline{\widehat{e_1}}) \mid \ldots \mid k_{\mathrm{n}}(\underline{\widehat{e_{\mathrm{n}}}})$$

When a refinement only has a single constructor, we may inline the definition to simplify presentation.

Example 7.2. Given $\widehat{\mathrm{Interval}}$ as the base domain, we can abstract values of data type

$$\textbf{data } \mathrm{intoption} = \mathrm{none}() \mid \mathrm{some}(\textbf{int})$$

as follows:

$$\alpha_{\widehat{\mathrm{DS}}}(\{\mathrm{none}(), \mathrm{some}(-2), \mathrm{some}(3)\}) = \mathrm{intoption\#m23}$$
$$\alpha_{\widehat{\mathrm{DS}}}(\{\mathrm{some}(41), \mathrm{some}(42)\}) = \mathrm{intoption\#4142}$$

with refinement definitions:

$$\textbf{refine } \mathrm{intoption\#m23} = \mathrm{none}() \mid \mathrm{some}([-2;3])$$
$$\textbf{refine } \mathrm{intoption\#4142} = \mathrm{some}([41;42])$$

We can abstract shape of inductive data types by using the least fixed point of our parameterized data shape domain, i.e., lfp $\widehat{\mathrm{X}}.\widehat{\mathrm{DataShape}}(\widehat{\mathrm{X}})$.

For example, we can abstract value of data type

$$\textbf{data } \mathrm{Nat} = \mathrm{zero}() \mid \mathrm{suc}(\mathrm{Nat})$$

as follows:

$$\alpha_{\widehat{\mathrm{DS}}}(\{\mathrm{zero}(), \mathrm{suc}(\mathrm{suc}(\mathrm{zero}()))\}) = \mathrm{Nat\#02}$$

with refinement definitions:

$$\textbf{refine } \mathrm{Nat\#02} = \mathrm{zero}() \mid \mathrm{suc}(\mathrm{Nat\#1})$$
$$\textbf{refine } \mathrm{Nat\#1} = \mathrm{suc}(\mathrm{Nat\#0}) \quad \textbf{refine } \mathrm{Nat\#0} = \mathrm{zero}()$$

▲

The lattice and widening operations on abstract elements follow directly from the refinement definitions, which we will describe afterwards.

$$\bot_{\widehat{\text{DS}}} \sqsubseteq_{\widehat{\text{DS}}} at\#r \sqsubseteq_{\widehat{\text{DS}}} \top_{\widehat{\text{DS}}} \qquad at\#r_1 \sqsubseteq_{\widehat{\text{DS}}} at\#r_2 \textbf{ iff } at\#r_1 \sqsubseteq_{\widehat{\text{R}}} at\#r_2$$
$$\bot \circledcirc_{\widehat{\text{DS}}} \widehat{d} = \widehat{d} \circledcirc_{\widehat{\text{DS}}} \bot = \widehat{d} \qquad \top \circledcirc_{\widehat{\text{DS}}} \widehat{d} = \widehat{d} \circledcirc_{\widehat{\text{DS}}} \top = \top$$
$$at_1\#r_1 \circledcirc_{\widehat{\text{DS}}} at_2\#r_2 = \text{if}(at_1 = at_2, at_1\#r_1 \circledcirc_{\widehat{\text{R}}} at_2\#r_2, \top) \quad \circledcirc \in \{\sqcup, \nabla\}$$
$$\bot \sqcap_{\widehat{\text{DS}}} \widehat{d} = \widehat{d} \sqcap_{\widehat{\text{DS}}} \bot = \bot \qquad \top \sqcap_{\widehat{\text{DS}}} \widehat{d} = \widehat{d} \sqcap_{\widehat{\text{DS}}} \top = \widehat{d}$$
$$at_1\#r_1 \sqcap_{\widehat{\text{DS}}} at_2\#r_2 = \text{if}(at_1 = at_2, at_1\#r_1 \sqcap_{\widehat{\text{R}}} at_2\#r_2, \bot)$$

Let the abstract content domain form a Galois connection with a concrete content domain $\wp(\text{E}) \underset{\alpha_{\widehat{\text{DS}}}}{\overset{\gamma_{\widehat{\text{DS}}}}{\leftrightarrows}} \widehat{\text{E}}$, then we can create abstract $\alpha_{\widehat{\text{DS}}}$ and concretization $\gamma_{\widehat{\text{DS}}}$ functions between

$$\text{Data} = \left\{ k(\underline{v}) \;\middle|\; \exists at.k(\underline{v}) \in [\![at]\!] \right\}$$

and $\widehat{\text{DataShape}}(\widehat{\text{E}})$ as follows:

$$\alpha_{\widehat{\text{DS}}}(ds) = \bigsqcup_{k(\underline{v}) \in ds} \text{if}(\forall i.v_i \in \text{E}, \alpha_{\widehat{\text{R}}}(\{k(\underline{v})\}), \top_{\widehat{\text{DS}}})$$
$$\gamma_{\widehat{\text{DS}}}(\bot_{\widehat{\text{DS}}}) = \emptyset \qquad \gamma_{\widehat{\text{DS}}}(\top_{\widehat{\text{DS}}}) = \text{Data} \qquad \gamma_{\widehat{\text{DS}}}(at\#r) = \gamma_{\widehat{\text{R}}}(at\#r)$$

Theorem 7.2. *Assuming a Galois connection for the leaf elements* $\wp(\text{E}) \underset{\alpha_{\widehat{\text{E}}}}{\overset{\gamma_{\widehat{\text{E}}}}{\leftrightarrows}} \widehat{\text{E}}$ *then we have a Galois connection for the data elements* $\wp(\text{Data}) \underset{\alpha_{\widehat{\text{DS}}}}{\overset{\gamma_{\widehat{\text{DS}}}}{\leftrightarrows}} \widehat{\text{DataShape}}(\widehat{\text{E}})$

Inductive Refinement Operations Recall that the data shape operations depended on the operations for inductive shape refinements, which we will now define. The lattice operations are based directly on the definitions given by Aiken and Murphy [1991].

Let $\Delta \subseteq \widehat{\text{Refinement}}(\widehat{\text{E}}) \times \widehat{\text{Refinement}}(\widehat{\text{E}})$ be a set of assumptions containing pairs of refinements $(at\#r_1, at\#r_2)$, such that $at\#r_1$ is assumed to be included in $at\#r_2$. Checking inclusion between two refinements is initially done in an empty set of assumptions, which is then possibly extended and passed down when checking recursive references.

We can then define inclusion using the following deductive rules:

$$\frac{(at\#r_1, at\#r_2) \in \Delta}{\Delta \vdash at\#r_1 \sqsubseteq_{\widehat{\mathrm{R}}} at\#r_2} \qquad \frac{\begin{array}{c} (at\#r_1, at\#r_2) \notin \Delta \\ \textbf{refine}\ at\#r_1 = \mathrm{k}_1(\underline{\widehat{e_1}}) \mid \cdots \mid \mathrm{k}_\mathrm{n}(\underline{\widehat{e_\mathrm{n}}}) \\ \textbf{refine}\ at\#r_2 = \mathrm{k}'_1(\underline{\widehat{e_1}'}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}'}) \\ \forall i \exists j. k_i = k'_j \wedge \underline{\Delta, (at\#r_1, at\#r_2) \vdash \widehat{e_i} \sqsubseteq_{\widehat{\mathrm{E}}} \widehat{e_j}'} \end{array}}{\Delta \vdash at\#r_1 \sqsubseteq_{\widehat{\mathrm{R}}} at\#r_2}$$

The inclusion rules state that either we already can determine the inclusion from the given set of assumptions, or otherwise we must check that $at\#r_2$ has at least the same constructors as in $at\#r_1$ and that the arguments of the same constructors are correspondingly included as well; note, how we extended the set of assumptions on the recursive call, which is required to detect inclusion for recursive references.

Example 7.3. The refinement Nat#1 is included in Nat#12, given the following definitions:

$$\begin{array}{c} \textbf{refine}\ \mathrm{Nat\#12} = \mathrm{suc}(\mathrm{Nat\#01}) \\ \textbf{refine}\ \mathrm{Nat\#01} = \mathrm{zero}() \mid \mathrm{suc}(\mathrm{Nat\#0}) \\ \textbf{refine}\ \mathrm{Nat\#1} = \mathrm{suc}(\mathrm{Nat\#0}) \\ \textbf{refine}\ \mathrm{Nat\#0} = \mathrm{zero}() \end{array}$$

We check inclusion using the following steps:

1. We start with $\vdash \mathrm{Nat\#1} \sqsubseteq_{\widehat{\mathrm{R}}} \mathrm{Nat\#12}$, and since the environment is empty we proceed to use the second rule, which for the sole suc constructor requires us to further check $(\mathrm{Nat\#1}, \mathrm{Nat\#12}) \vdash \mathrm{Nat\#0} \sqsubseteq_{\widehat{\mathrm{DS}}} \mathrm{Nat\#01}$

2. By definition, checking a data shape requires checking the corresponding refinement $(\mathrm{Nat\#1}, \mathrm{Nat\#12}) \vdash \mathrm{Nat\#0} \sqsubseteq_{\widehat{\mathrm{R}}} \mathrm{Nat\#01}$

3. Checking $(\mathrm{Nat\#1}, \mathrm{Nat\#12}) \vdash \mathrm{Nat\#0} \sqsubseteq_{\widehat{\mathrm{R}}} \mathrm{Nat\#01}$ also requires using the second rule, where we can see that Nat#01 contains the sole zero() constructor of Nat#0, and because it has no arguments we have fulfilled all required assumptions.

▲

The least upper bound and greatest lower bound operations require a memoization map $\Phi \in \widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}}) \times \widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}}) \to \widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}})$ which keeps track of results for already merged refinements. We define the least upper bound using the following rules:

$$\frac{\Phi(at\#r_1, at\#r_2) = at\#r_3}{\Phi \vdash at\#r_1 \sqcup_{\widehat{\mathrm{R}}} at\#r_2 = at\#r_3}$$

$$\frac{\begin{array}{c} (at\#r_1, at\#r_2) \notin \mathrm{dom}\ \Phi \\ \textbf{refine}\ at\#r_1 = \mathrm{k}_1(\underline{\widehat{e_1}}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}}) \mid \mathrm{k}'_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}}'}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}'}) \\ \textbf{refine}\ at\#r_2 = \mathrm{k}_1(\underline{\widehat{e_1}''}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}''}) \mid \mathrm{k}''_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}}''}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}''}) \\ \forall ij.k'_i \neq k''_j \qquad \underline{\forall i.\Phi[(at\#r_1, at\#r_2) \mapsto at\#r_3] \vdash \widehat{e}_i \sqcup_{\widehat{\mathrm{E}}} \widehat{e}''_i = \widehat{e}'''_i} \\ \textbf{refine}\ at\#r_3 = \mathrm{k}_1(\underline{\widehat{e_1}'''}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}'''}) \mid \mathrm{k}'_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}}'}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}'}) \\ \mid \mathrm{k}'_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}}'}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}''}) \end{array}}{\Phi \vdash at\#r_1 \sqcup_{\widehat{\mathrm{R}}} at\#r_2 = at\#r_3}$$

This states that the upper bound is either the one given by the memoization, or a refinement $at\#r_3$ which contains the constructors of both $at\#r_1$ and $at\#r_2$ and where the arguments for the common constructors are merged using the least upper bound with an updated memoization. In practice, if the definition of $at\#r_3$ is not already existing in our considered set of refinements then we will create it.

Example 7.4. We will calculate the least upper bound of Nat#1 and Nat#2 given the following existing definitions:

$$\begin{gathered} \textbf{refine}\ \mathrm{Nat\#2} = \mathrm{suc}(\mathrm{Nat\#1}) \\ \textbf{refine}\ \mathrm{Nat\#1} = \mathrm{suc}(\mathrm{Nat\#0}) \\ \textbf{refine}\ \mathrm{Nat\#0} = \mathrm{zero}() \end{gathered}$$

The calculation is done using the following steps:

- We first need to compute $[] \vdash \mathrm{Nat\#1} \sqcup_{\widehat{\mathrm{R}}} \mathrm{Nat\#2} = \mathrm{Nat\#12}$, which requires uses the second rule to create a refinement:

  $$\textbf{refine}\ \mathrm{Nat\#12} = \mathrm{suc}(\mathrm{Nat\#01})$$

  which also requires us to calculate: $[(\mathrm{Nat\#1} \sqcup_{\widehat{\mathrm{R}}} \mathrm{Nat\#2}) \mapsto \mathrm{Nat\#12}] \vdash \mathrm{Nat\#0} \sqcup_{\widehat{\mathrm{DS}}} \mathrm{Nat\#1} = \mathrm{Nat\#01}$

- By definition, this means that we need to calculate $[(\mathrm{Nat\#1} \sqcup_{\widehat{\mathrm{R}}} \mathrm{Nat\#2}) \mapsto \mathrm{Nat\#12}] \vdash \mathrm{Nat\#0} \sqcup_{\widehat{\mathrm{R}}} \mathrm{Nat\#1} = \mathrm{Nat\#01}$, which we do using the second rule to get refinement:

$$\textbf{refine}\ \mathrm{Nat\#01} = \mathrm{zero} \mid \mathrm{suc}(\mathrm{Nat\#0})$$

- Since there was no common constructor, there are no more components to merge, finishing the least upper bound calculation.

▲

The greatest lower bound operator on refinements is defined similarly as:

$$\frac{\Phi(at\#r_1, at\#r_2) = at\#r_3}{\Phi \vdash at\#r_1 \sqcap_{\widehat{\mathrm{R}}} at\#r_2 = at\#r_3}$$

$$\frac{\begin{array}{c} (at\#r_1, at\#r_2) \notin \mathrm{dom}\ \Phi \\ \textbf{refine}\ at\#r_1 = \mathrm{k}_1(\underline{\widehat{e_1}}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}}) \mid \mathrm{k}'_{\mathrm{n+1}}(\underline{\widehat{e_{\mathrm{n+1}}}'}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}'}) \\ \textbf{refine}\ at\#r_2 = \mathrm{k}_1(\underline{\widehat{e_1}''}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}''}) \mid \mathrm{k}''_{\mathrm{n+1}}(\underline{\widehat{e_{\mathrm{n+1}}}''}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}''}) \\ \forall i, j. k'_i \neq k''_j \qquad \dfrac{\forall i. \Phi[(at\#r_1, at\#r_2) \mapsto at\#r_3] \vdash \widehat{e}_i \sqcap_{\widehat{\mathrm{E}}} \widehat{e}''_i = \widehat{e}'''_i}{\textbf{refine}\ at\#r_3 = \mathrm{k}_1(\underline{\widehat{e_1}'''}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}'''})} \end{array}}{\Phi \vdash at\#r_1 \sqcap_{\widehat{\mathrm{R}}} at\#r_2 = at\#r_3}$$

The main difference from the least upper bound is that the result refinement only has common constructors and that constructor arguments are recursively merged using greatest lower bounds as required.

The widening operator is more elaborate since it needs to infer inductive rules in order to ensure that calling programs terminate. Our widening operator is an extension of the definition given in Cousot and Cousot [1995], and in particular it may need to transform the existing refinement definitions[3]. The procedure can be written down fully formally, but it quickly becomes incomprehensible. Instead we will describe the procedure in a step-by-step fashion:

1. Assume we want to widen the two refinements $at\#r_1$ and $at\#r_2$ to a refinement $at\#r_3$ given the following definitions (so the constructors k' and the constructors k'' are disjoint):

$$\textbf{refine}\ at\#r_1 = \mathrm{k}_1(\underline{\widehat{e_1}}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}}) \mid \mathrm{k}'_{\mathrm{n+1}}(\underline{\widehat{e_{\mathrm{n+1}}}'}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}'})$$
$$\textbf{refine}\ at\#r_2 = \mathrm{k}_1(\underline{\widehat{e_1}''}) \mid \cdots \mid \mathrm{k_n}(\underline{\widehat{e_\mathrm{n}}''}) \mid \mathrm{k}''_{\mathrm{n+1}}(\underline{\widehat{e_{\mathrm{n+1}}}''}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}}''})$$

[3]To avoid breaking other abstract values, we in practice perform the transformation on a local copy of the rule set and then add back the local refinement definitions that are relevant for the result

2. The first merging step is to create a refinement definition for $at\#r_3$ that contains both sets of constructor definitions:

$$\begin{aligned}\textbf{refine }at\#r_3 = {} & \mathrm{k}_1(\underline{\widehat{e_1}}) \mid \cdots \mid \mathrm{k}_\mathrm{n}(\underline{\widehat{e_\mathrm{n}}}) \mid \mathrm{k}'_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}'}}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}'}}) \\ & \mid \mathrm{k}_1(\underline{\widehat{e_1}''}) \mid \cdots \mid \mathrm{k}_\mathrm{n}(\underline{\widehat{e_\mathrm{n}}''}) \mid \mathrm{k}''_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}''}}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}''}})\end{aligned}$$

3. This may have multiple alternative definitions for the same constructor if there were any in common, so the second step is to recursively merge the arguments of the common constructors.

4. On refinements that should be merged recursively, we will this time not continue eagerly creating new definitions but instead update the memoization environment Φ, so that the refinements to-be-merged map to fresh target refinement names. For example, if we need to recursively merge refinements $at\#r_4$ and $at\#r_5$ we produce a fresh refinement $at\#r_6$ that is used as the new value (without a definition) along with environment $\Phi[(at\#r_4, at\#r_5) \mapsto at\#r_6]$.

5. If the definition of $at\#_1$ (or $at\#_2$) has more the choice between more than one constructor then we replace all references to it in the existing refinement definitions and memoization environment Φ with $at\#_3$.

6. We will then transform the definition for $at\#r_3$ to use the merged result arguments $\underline{\widehat{e_i}'''}$, so we get:

$$\begin{aligned}\textbf{refine }at\#r_3 = {} & \mathrm{k}_1(\underline{\widehat{e_1}'''}) \mid \cdots \mid \mathrm{k}_\mathrm{n}(\underline{\widehat{e_\mathrm{n}}'''}) \mid \mathrm{k}'_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}'}}) \mid \cdots \mid \mathrm{k}'_\mathrm{m}(\underline{\widehat{e_\mathrm{m}'}}) \\ & \mid \mathrm{k}''_{\mathrm{n}+1}(\underline{\widehat{e_{\mathrm{n}+1}''}}) \mid \cdots \mid \mathrm{k}''_\mathrm{m}(\underline{\widehat{e_\mathrm{m}''}})\end{aligned}$$

7. Finally, if Φ is not empty, then we repeat the same widening procedure to a new pair $(at\#r'_1, at\#r'_2) \in \operatorname{dom} \Phi$, merging it to the target refinement $\Phi(at\#r'_1, at\#r'_2)$.

Example 7.5. We will calculate the widening of Nat#12 and Nat#123 to Nat#s, given definitions

$$\begin{gathered}\textbf{refine } \mathrm{Nat\#12} = \mathrm{suc}(\mathrm{Nat\#01}) \\ \textbf{refine } \mathrm{Nat\#123} = \mathrm{suc}(\mathrm{Nat\#012}) \\ \textbf{refine } \mathrm{Nat\#01} = \mathrm{zero}() \mid \mathrm{suc}(\mathrm{Nat\#0}) \\ \textbf{refine } \mathrm{Nat\#012} = \mathrm{zero}() \mid \mathrm{suc}(\mathrm{Nat\#01}) \\ \textbf{refine } \mathrm{Nat\#0} = \mathrm{zero}()\end{gathered}$$

We proceed by following the steps of the widening procedure:

- The first step is to create the definition for Nat#s:

$$\textbf{refine}\ \text{Nat\#s} = \text{suc}(\text{Nat\#01}) \mid \text{suc}(\text{Nat\#012})$$

- Because there was a common constructor suc, we proceed by widening the constructor arguments $\text{Nat\#01} \nabla_{\widehat{\text{DS}}} \text{Nat\#012}$.

- Since by definition the constructor arguments require recursively widening two refinements, we create a fresh refinement Nat#r as required and update the memoization map so we get $[(\text{Nat\#01}, \text{Nat\#012}) \mapsto \text{Nat\#r}]$

- We then transform the definition of Nat#s to get:

$$\textbf{refine}\ \text{Nat\#s} = \text{suc}(\text{Nat\#r})$$

- Since both Nat#12 and Nat#123 only had a single constructor, there is no replacement with Nat#s necessary in the rest of the definitions.

- We now continue merging Nat#01 and Nat#012 to Nat#r, by creating the definition:

$$\textbf{refine}\ \text{Nat\#r} = \text{zero}() \mid \text{suc}(\text{Nat\#0}) \mid \text{suc}(\text{Nat\#01})$$

- Because there again was a common constructor suc, we need to merge recursively merge suc(Nat#0) and suc(Nat#01) to fresh refinement Nat#p, getting new memoization environment $[(\text{Nat\#0}, \text{Nat\#01}) \mapsto \text{Nat\#p}]$ and transformed definition:

$$\textbf{refine}\ \text{Nat\#r} = \text{zero}() \mid \text{suc}(\text{Nat\#p})$$

- Since Nat#01 had more than one constructor, we replace all occurrences of Nat#01 with Nat#r in the memoization environment getting $[(\text{Nat\#0}, \text{Nat\#r}) \mapsto \text{Nat\#p}]$ and similarly transform the definitions to:

$$\textbf{refine}\ \text{Nat\#s} = \text{suc}(\text{Nat\#r})$$
$$\textbf{refine}\ \text{Nat\#r} = \text{zero}() \mid \text{suc}(\text{Nat\#p})$$
$$\textbf{refine}\ \text{Nat\#0} = \text{zero}()$$

  where we dropped definitions irrelevant for the rest of the widening.

- We proceed create a definition for Nat#p, merging Nat#0 and Nat#r to get:

$$\textbf{refine}\ \text{Nat\#p} = \text{zero}() \mid \text{suc}(\text{Nat\#p})$$

- There are no duplicate constructors, and since Nat#r had more than one possible constructor, we need to replace it in the definition with Nat#p, to get the following set of refinement definitions:

$$\textbf{refine}\ \text{Nat\#s} = \text{suc}(\text{Nat\#p})$$
$$\textbf{refine}\ \text{Nat\#p} = \text{zero}() \mid \text{suc}(\text{Nat\#p})$$

- Here we can see that Nat#p is simply the Nat datatype, so we replace it and get the following final refinements:

$$\textbf{refine}\ \text{Nat\#s} = \text{suc}(\text{Nat})$$

▲

Given a Galois connection $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$, we can define abstraction $\alpha_{\widehat{\mathrm{R}}}$ and concretization $\gamma_{\widehat{\mathrm{R}}}$ functions between

$$\mathrm{PData}(\mathrm{E}) = \left\{ k(\underline{v}) \;\middle|\; k(\underline{v}) \in \mathrm{Data} \wedge \forall i.v_i \in \mathrm{E} \right\}$$

and refinements $\widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}})$ as follows:

$$\alpha_{\widehat{\mathrm{R}}}(ds) = \bigsqcup_{k(\underline{v}) \in ds} at\#rk \qquad (\textbf{where refine}\ at\#rk = k(\underline{\alpha_{\widehat{\mathrm{E}}}(\{v\})}))$$
$$\gamma_{\widehat{\mathrm{R}}}(at\#r) = \left\{ k_i(\underline{v_i}) \;\middle|\; i \in [1;\mathrm{n}] \wedge \underline{v_i \in \gamma_{\widehat{\mathrm{E}}}(e_i)} \wedge k_i(\underline{v_i}) \in [\![at]\!] \right\}$$
$$(\textbf{where refine}\ at\#r = k_1(\underline{\widehat{e_1}}) \mid \cdots \mid k_{\mathrm{n}}(\underline{\widehat{e_{\mathrm{n}}}}))$$

Theorem 7.3. *Given a Galois connection* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$*, then there is a Galois connection* $\mathrm{PData}(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{R}}}}{\overset{\gamma_{\widehat{\mathrm{R}}}}{\leftrightarrows}} \widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}})$

Choice Domain

To allow a choice between multiple data-types we use the classical choice domain [Scott, 1976; Winskel, 1993a] enriched with top, defined as follows:

$$ea \in \bigoplus(\underline{A}) ::= \bot_\oplus \mid \top_\oplus \mid \mathrm{in}_i(a_i), \quad i \in \mathbb{N}, a_i \in A_i$$

Given that injections are exclusive $\mathrm{in}_i(a) \neq \mathrm{in}_j(b)$ for $i \neq j$, then the ordering is:

$$\begin{gathered}\bot_\oplus \sqsubseteq_\oplus \mathrm{in}_i(a) \sqsubseteq_\oplus \top_\oplus \\ \mathrm{in}_i(a) \sqsubseteq_\oplus \mathrm{in}_i(a') \textbf{ iff } a \sqsubseteq_{\mathrm{A}_i} a'\end{gathered}$$

The least upper-bound is:

$$\begin{array}{ll}\bot_\oplus \sqcup_\oplus ea = ea \sqcup_\oplus \bot = ea & \top_\oplus \sqcup_\oplus ea = ea \sqcup_\oplus \top_\oplus = \top_\oplus \\ \mathrm{in}_i(a) \sqcup_\oplus \mathrm{in}_i(a') = \mathrm{in}_i(a \sqcup_{\widehat{\mathrm{A}}_i} a') & \mathrm{in}_j(a) \sqcup_\oplus \mathrm{in}_i(b) = \top_\oplus \textbf{ iff } i \neq j\end{array}$$

Given a Galois connection for each option in the choice $\wp(\mathrm{A}_i) \xleftrightarrow[\alpha_{\widehat{\mathrm{A}}_i}]{\gamma_{\widehat{\mathrm{A}}_i}} \widehat{\mathrm{A}}_i$, we can define abstraction $\alpha_\oplus$ and concretization $\gamma_\oplus$ functions between the powerset of disjoint union of concrete domains $\wp\left(\biguplus \underline{A}\right)$ and the choice domain $\bigoplus(\underline{\widehat{\mathrm{A}}})$ as follows:

$$\begin{gathered}\alpha_\oplus(eas) = \bigsqcup_{\mathrm{in}_i(a) \in eas} \mathrm{in}_i(\alpha_{\widehat{\mathrm{A}}_i}(\{a\})) \\ \gamma_\oplus(\bot_\oplus) = \emptyset \quad \gamma_\oplus(\mathrm{in}_i(\widehat{a})) = \{\mathrm{in}_i(\widehat{a}) \mid a \in \gamma_\oplus(\widehat{a})\} \\ \gamma_\oplus(\top_\oplus) = \biguplus \underline{A}\end{gathered}$$

Theorem 7.4. *Given* $\forall i \left(\wp(\mathrm{A}_i) \xleftrightarrow[\alpha_{\widehat{\mathrm{A}}_i}]{\gamma_{\widehat{\mathrm{A}}_i}} \widehat{\mathrm{A}}_i\right)$ *then* $\wp\left(\biguplus \underline{\mathrm{A}}\right) \xleftrightarrow[\alpha_\oplus]{\gamma_\oplus} \bigoplus(\underline{\widehat{\mathrm{A}}})$

Fixedpoint Domain

For any parameterized domain $\forall$E.F(E), we can apply the least fixed-point to get an inductive domain FX = lfp X.F(X) [Scott, 1976; Smyth and Plotkin, 1982; Winskel, 1993b]. If we additionally have that for all domains E that form a lattice then F(E) forms a lattice, then we get that the fixedpoint domain FX also forms a lattice.

Now let an abstract base domain form a Galois connection $\wp\,(\mathrm{F}(\mathrm{E})) \xrightleftharpoons[\alpha_{\mathrm{F}}]{\gamma_{\mathrm{F}}} \widehat{\mathrm{F}}(\widehat{\mathrm{E}})$ for any abstract base element domain $\widehat{\mathrm{E}}$ with Galois connection $\wp\,(\mathrm{E}) \xrightleftharpoons[\alpha_{\mathrm{E}}]{\gamma_{\mathrm{E}}} \widehat{\mathrm{E}}$, then we straightforwardly have the following abstraction α_{Fix} and concretization $\gamma_{\widehat{\mathrm{Fix}}}$ functions between $\wp(\mathrm{lfp}\ \mathrm{X.F(X)})$ and $\wp(\mathrm{lfp}\ \widehat{\mathrm{X}}.\widehat{\mathrm{F}}(\widehat{\mathrm{X}}))$:

$$\alpha_{\mathrm{Fix}}(fs) = \alpha_{\mathrm{F}}(fs) \qquad \gamma_{\widehat{\mathrm{Fix}}}(\widehat{f}) = \gamma_{\widehat{\mathrm{F}}}(\widehat{f})$$

Theorem 7.5. *If that for continuous parametrized domains* F, $\widehat{\mathrm{F}}$ *it holds that for all* E, $\widehat{\mathrm{E}}$ *such that* $\wp\,(\mathrm{E}) \xrightleftharpoons[\alpha_{\widehat{\mathrm{E}}}]{\gamma_{\widehat{\mathrm{E}}}} \widehat{\mathrm{E}}$ *then* $\wp\,(\mathrm{F}(\mathrm{E})) \xrightleftharpoons[\alpha_{\widehat{\mathrm{F}}}]{\gamma_{\widehat{\mathrm{F}}}} \widehat{\mathrm{F}}(\widehat{\mathrm{E}})$*, then* $\wp\,(\mathrm{lfp}\ \mathrm{X.F(X)}) \xrightleftharpoons[\alpha_{\widehat{\mathrm{Fix}}}]{\gamma_{\widehat{\mathrm{Fix}}}} \mathrm{lfp}\ \widehat{\mathrm{X}}.\widehat{\mathrm{F}}(\widehat{\mathrm{X}})$

Value Domains

We presented the required components for abstracting abstractions, and now all that is left is putting everything together using the fixedpoint domain.

We first define a parameterized value shape domain that combine all our different shape domains together:

$$\widehat{\mathrm{PValueShape}}(\widehat{\mathrm{E}}) = \widehat{\mathrm{Interval}} \oplus \widehat{\mathrm{SetShape}}(\widehat{\mathrm{E}}) \oplus \widehat{\mathrm{DataShape}}(\widehat{\mathrm{E}})$$

We define a corresponding parameterized concrete domain:

$$\mathrm{PValue(E)} = \mathbb{Z} \uplus \wp\,(E) \uplus \mathrm{Data}$$

Corollary 7.1. *If given a Galois connection for the parameter element domain* $\wp\,(\mathrm{E}) \xrightleftharpoons[\alpha_{\widehat{\mathrm{E}}}]{\gamma_{\widehat{\mathrm{E}}}} \widehat{\mathrm{E}}$*, then we have a Galois connection for the parameterized value domain* $\wp\,(\mathrm{PValue(E)}) \xrightleftharpoons[\alpha_{\widehat{\mathrm{PVS}}}]{\gamma_{\widehat{\mathrm{PVS}}}} \widehat{\mathrm{PValueShape}}(\widehat{\mathrm{E}})$

Finally, we define our complete value shape abstract domain using the least fixedpoint domain and the parameterized value shape domain: $\widehat{\mathrm{ValueShape}} = \mathrm{lfp}\ \widehat{\mathrm{X}}.\widehat{\mathrm{PValueShape}}(\widehat{\mathrm{X}})$. Similarly, we have the equivalence $\mathrm{Value} = \mathrm{lfp}\ \mathrm{X.PValue(X)}$.

Corollary 7.2. *We have a Galois connection* $\wp\,(\mathrm{Value}) \xrightleftharpoons[\alpha_{\widehat{\mathrm{VS}}}]{\gamma_{\widehat{\mathrm{VS}}}} \widehat{\mathrm{ValueShape}}$

Abstract State Domains

Given our shape and relational domains, we can construct abstractions for our stores and results (control operators) which represent the state that Rascal programs manipulate.

Abstract Store Domain Our abstract store maps variables to a pair consisting of a Boolean indicating whether the variable is assigned and the target abstract value:

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

We lift our orderings and lattice operations from the value shape domain to abstract stores (resp. $\sqsubseteq_{\widehat{\text{Store}}}$, $\sqcup_{\widehat{\text{Store}}}$, $\sqcap_{\widehat{\text{Store}}}$, $\nabla_{\widehat{\text{Store}}}$). We define abstraction $\alpha_{\widehat{\text{Store}}}$ and concretization $\gamma_{\widehat{\text{Store}}}$ operations between concrete sets of stores $\wp(\text{Store})$ and abstract stores $\widehat{\text{Store}}$:

$$\alpha_{\widehat{\text{Store}}}(\Sigma) = \bigsqcup_{\sigma \in \Sigma} \lambda x.\text{if}(x \in \textbf{dom}\,\sigma, (\text{ff}, \alpha_{\widehat{\text{V}}}(\{\sigma(x)\})), (\text{tt}, \bot_{\widehat{\text{VS}}}))$$

$$\gamma_{\widehat{\text{Store}}}(\widehat{\sigma}) = \left\{\sigma \mid \forall x.\widehat{\sigma}(x) = (b, \widehat{vs}) \land \text{if}(x \in \textbf{dom}\,\sigma, \sigma(x) \in \gamma_{\widehat{\text{V}}}(\widehat{vs}), b)\right\}$$

Theorem 7.6. *We have a Galois connection* $\wp(\text{Store}) \underset{\alpha_{\widehat{\text{Store}}}}{\overset{\gamma_{\widehat{\text{Store}}}}{\leftrightarrows}} \widehat{\text{Store}}$

Abstract Result Domain A particularly interesting side-effect of Schmidt-style abstract interpretation is that it enables to handle abstraction of control flow more directly. Traditionally, control flow is often handled using a denotational collecting semantics or by explicitly constructing a control flow graph, but both these approaches are non-trivial to apply for a rich language like Rascal.

Our idea is to model different kind of results—success, failure, errors—that affect control flow directly in a $\widehat{\text{ResultSet}}$ domain which keeps track of possible results for a particular execution case, in addition to managing a separate store for each possible result kind. Keeping separate stores is cheap, since there are only three kinds of results, and important to maintain precision, since different kind of results lead to different paths in concrete programs.

$$\widehat{vres} \in \widehat{\text{ValueResult}} ::= \textbf{success}\ \widehat{v} \mid \widehat{exres}$$

$$\widehat{exres} \in \widehat{\text{ExceptionalResult}} ::= \textbf{fail} \mid \textbf{error}$$

$$\widehat{res} \in \widehat{\text{Result}} ::= \cdot \mid \widehat{vres}; \widehat{\sigma} \qquad \widehat{Res} \in \widehat{\text{ResultSet}} ::= \underline{\widehat{res}}$$

The ordering of result sets checks is dependent on the presence of result kinds and the ordering of contained values and stores. Formally, we can define it as follows:

$$\widehat{Res_1} \sqsubseteq_{\widehat{\mathrm{RS}}} \widehat{Res_2}$$
$$\textbf{iff}$$
$$(\textbf{success}\ v_1; \sigma_1) \in \widehat{Res_1} \Rightarrow \begin{array}{r} \exists v_2, \sigma_2 . v_1 \sqsubseteq_{\widehat{\mathrm{VS}}} v_2 \wedge \sigma_1 \sqsubseteq_{\widehat{\mathrm{Store}}} \sigma_2 \wedge \\ (\textbf{success}\ v_2; \sigma_2) \in \widehat{Res_2} \end{array}$$
$$\wedge$$
$$(\widehat{exres}; \widehat{\sigma}_1) \in \widehat{Res_1} \Rightarrow \exists \widehat{\sigma}_2 . \widehat{\sigma}_1 \sqsubseteq_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2 \wedge (\widehat{exres}; \widehat{\sigma}_2) \in \widehat{Res_2}$$

The least upper bound collects the different kinds of results available in both input result sets, merging the components—result value, store—of the common ones. We formally define the least upper bound as follows (widening analogous):

$$\widehat{Res_1} \sqcup_{\widehat{\mathrm{RS}}} \widehat{Res_2} = \mathrm{merge}(\widehat{Res_1}, \widehat{Res_2})$$

$$\mathrm{merge}(\widehat{Res}) = \begin{cases} (\textbf{success}\ (\widehat{v}_1 \sqcup_{\widehat{\mathrm{VS}}} \widehat{v}_2); \widehat{\sigma}_1 \sqcup_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2), \mathrm{merge}(\widehat{Res}') \\ \qquad \text{if } \widehat{Res} = (\textbf{success}\ \widehat{v}_1; \widehat{\sigma}_1), (\textbf{success}\ \widehat{v}_2; \widehat{\sigma}_2), \widehat{Res}' \\ (\textbf{fail}; \widehat{\sigma}_1 \sqcup_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2), \mathrm{merge}(\widehat{Res}') \\ \qquad \text{if } \widehat{Res} = (\textbf{fail}; \widehat{\sigma}_1), (\textbf{fail}; \widehat{\sigma}_2), \widehat{Res}' \\ (\textbf{error}; \widehat{\sigma}_1 \sqcup_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2), \mathrm{merge}(\widehat{Res}') \\ \qquad \text{if } \widehat{Res} = (\textbf{error}; \widehat{\sigma}_1), (\textbf{error}; \widehat{\sigma}_2), \widehat{Res}' \\ \widehat{Res} \qquad \text{otherwise} \end{cases}$$

The greatest lower bound contains the result kinds that were present in both input results sets, and then calculates the greatest lower bound of their components. Formally, we define the greatest lower bound as follows:

$$\widehat{Res_1} \sqcap_{\widehat{\mathrm{RS}}} \widehat{Res_2} = \bigsqcup_{\substack{\widehat{res_1} \in \widehat{Res_1} \wedge \\ \widehat{res_2} \in \widehat{Res_2}}} (\widehat{res_1} \sqcap_{\widehat{\mathrm{Rs}}} \widehat{res_2})$$

$$\widehat{res_1} \sqcap_{\widehat{\mathrm{Rs}}} \widehat{res_2} = \begin{cases} \textbf{success}\ (\widehat{v}_1 \sqcap_{\widehat{\mathrm{VS}}} \widehat{v}_2); \widehat{\sigma}_1 \sqcap_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2 & \text{if } \widehat{res_i} = \textbf{success}\ \widehat{v}_i; \widehat{\sigma}_i \\ \textbf{fail}; \widehat{\sigma}_1 \sqcap_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2 & \text{if } \widehat{res_i} = \textbf{fail}; \widehat{\sigma}_i \\ \textbf{error}; \widehat{\sigma}_1 \sqcap_{\widehat{\mathrm{Store}}} \widehat{\sigma}_2 & \text{if } \widehat{res_i} = \textbf{error}; \widehat{\sigma}_i \\ \varepsilon & \text{otherwise} \end{cases}$$

We define abstraction $\alpha_{\widehat{\text{RS}}}$ and concretization $\gamma_{\widehat{\text{RS}}}$ operations between the concrete domain $\wp(\text{ValueResult} \times \text{Store})$ and abstract result sets $\widehat{\text{ResultSet}}$ as follows:

$$\alpha_{\widehat{\text{RS}}}(Res) = \bigsqcup_{(vres,\sigma)\in Res} \alpha_{\widehat{\text{VR}}}(\{vres\}); \alpha_{\widehat{\text{Store}}}(\{\sigma\})$$

$$\alpha_{\widehat{\text{VR}}}(Vres) = \bigsqcup_{\textbf{value}\ v\in Vres} \left(\textbf{value}\ \alpha_{\widehat{\text{VS}}}(v)\right) \sqcup \bigsqcup_{\widehat{exres}\in Vres} \widehat{exres}$$

$$\gamma_{\widehat{\text{RS}}}(\underline{\widehat{vres}; \widehat{\sigma}}) = \bigcup_i \left\{(vres, \sigma) \mid vres \in \gamma_{\widehat{\text{VR}}}(\widehat{vres}_i) \wedge \sigma \in \gamma_{\widehat{\text{St}}}(\widehat{\sigma}_i)\right\}$$

$$\gamma_{\widehat{\text{VR}}}(\textbf{value}\ \widehat{vs}) = \left\{\textbf{value}\ v \mid v \in \gamma_{\widehat{\text{VS}}}(\widehat{vs})\right\} \qquad \gamma_{\widehat{\text{VR}}}(\widehat{exres}) = \{\widehat{exres}\}$$

Theorem 7.7. *We have a Galois connection*

$$\wp(\text{ValueResult} \times \text{Store}) \underset{\alpha_{\widehat{\text{RS}}}}{\overset{\gamma_{\widehat{\text{RS}}}}{\leftrightarrows}} \widehat{\text{ResultSet}}$$

7.4 Abstract Semantics

The genius of Schmidt-style abstract interpretation is that it allows us to get abstract operational rules in a way that closely corresponds to our existing concrete operational rules. The operational structure of the rules can be derived almost mechanically [Schmidt, 1998; Bodin et al., 2015], which we exemplify by translation of the operational rules for the Rascal UltraLight subset. The creative work is therefore providing abstract definitions for conditions and semantic operations such as pattern matching, and defining *trace memoization strategies* for non-structurally recursive operational rules in order to make it possible to finitely approximate an infinite number of concrete traces and produce a terminating static analysis.

Abstracting Operational Rules

In Schmidt-style abstract interpretation, we have to for each concrete judgment create an over-approximative abstract judgment that captures the result for a set of possible traces. Recall, that the concrete operational rules for expressions have a judgment of form $e; \sigma \underset{\text{expr}}{\Longrightarrow} vres; \sigma'$ which states that evaluating an expression e in store σ produces the value result $vres$ and store σ'. The corresponding abstract semantics judgment has form $e; \widehat{\sigma} \underset{\text{a-expr}}{\Longrightarrow} \widehat{Res}$ which evaluates expression e in store σ producing a result set $\widehat{Res} = (\widehat{vres}_1; \widehat{\sigma}_1), \ldots, (\widehat{vres}_{\mathrm{n}}; \widehat{\sigma}_{\mathrm{n}})$ consisting of pairs of possible abstract value results and corresponding abstract stores. These sets of pairs are meant to finitely over-approximate the possibly infinite sets of concrete results from evaluating e by providing abstractions of possible values and stores for each different kind of result (success, failure, errors).

Collecting premise notation In concrete semantics we usually only require that premises are satisfied by a single rule, but in abstract semantics we have to consider all possible applicable rules to preserve soundness. Thus, to reason about the generalized sets of results we introduce the notation

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

to collect all derivations with input i into a single output O that is equal to the least upper bound of the output of each individual derivation o.

Translating semantics To illustrate the steps required in Schmidt-style translation of operational rules, we will use the operational rules for variable accesses. Recall that 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.

$$\text{E-V-S}\frac{x \in \operatorname{dom} \sigma}{x; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{value } \sigma(x); \sigma} \qquad \text{E-V-E{\scriptsize R}}\frac{x \notin \operatorname{dom} \sigma}{x; \sigma \underset{\text{expr}}{\Longrightarrow} \textbf{error}; \sigma}$$

To translate the concrete rules for a syntactic element to abstract operational rules, we follow three steps in general:

1. For each concrete rule, create an abstract rule that uses an judgment for evaluation of a concrete syntactic form, e.g., variables.
2. Replace the concrete conditions and semantic operations with the equivalent abstract conditions and semantic operations for target abstract values.
3. Create a general abstract rule that collects all possible evaluations of the syntax-specific judgment rules, over-approximating all possible result values.

The translated abstract operational rules for evaluating variable accesses, are presented below:

$$\text{AE-V-S}\frac{\widehat{\sigma}(x) = (b, \widehat{vs})}{x; \widehat{\sigma} \xRightarrow[\text{a-expr-v}]{} \textbf{value } \widehat{vs}; \widehat{\sigma}} \qquad \text{AE-V-ER}\frac{\widehat{\sigma}(x) = (\text{tt}, \widehat{vs})}{x; \widehat{\sigma} \xRightarrow[\text{a-expr-v}]{} \textbf{error}; \widehat{\sigma}}$$

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

In the first step, we create two abstract syntax-specific rules AE-V-S and AE-V-Er corresponding to E-V-S and E-V-Er respectively. In the second step, we must translate the conditions and semantic operations which for our example requires:

- Translating the test $x \in \text{dom } \sigma$ with a corresponding condition on $\widehat{\sigma}$. Recall that in our abstract store, we have assigned a pair $(b, \widehat{vs})$ for each variable x, where the first component b is a Boolean that indicates whether the variable x is potentially unassigned; thus, we look at this component to decide whether we have $x \notin \text{dom } \sigma$ (if $b = \text{tt}$) and otherwise we may only have $x \in \text{dom } \sigma$.
- Translating the lookup $\sigma(x)$ in the result of E-V-S to a lookup of values in $\widehat{\sigma}$, which requires extracting the second component assigned to variable x.

Finally, in the last step we use abstract rule AE-V to collects the two syntax-specific abstract operational rules AE-V-S and AE-V-Er.

The possible shapes of the result value depend on the pair assigned to x in the abstract store; if the resulting value is $\perp$, we propagate the emptiness dropping the complete result value. We can use the following examples, to illustrate the possible outcome result shapes:

Store	**Value Results**	**Rules**
$\widehat{\sigma}[x \mapsto (\text{ff}, \perp_{\widehat{\text{VS}}})]$	$\emptyset$	AE-V-S
$\widehat{\sigma}[x \mapsto (\text{ff}, [1;3])]$	**value** $[1;3]$	AE-V-S
$\widehat{\sigma}[x \mapsto (\text{tt}, [1;3])]$	$\{\textbf{value } [1;3], \textbf{error}\}$	AE-V-S, AE-V-Er
$\widehat{\sigma}[x \mapsto (\text{tt}, \perp_{\widehat{\text{VS}}})]$	**error**	AE-V-S, AE-V-Er

The abstract operational semantics rules for the rest of the basic expressions (excluding visitors) are presented in Figure 7.2, Figure 7.3 and

Figure 7.4. The rules are generally translated from the corresponding concrete semantics rules in Chapter 6, using the steps we presented; note that sequencing expressions are a special case of block expressions which have no variable declarations and only two expressions in the body. There are a few things to note regarding the abstract conditions and semantic operations:

- When the concrete semantics checks that a result of evaluating an expression has a particular shape, the abstract semantics checks whether the particular shape is possibly included in the result set of evaluating the same expression. E.g., to check that evaluation of e has succeeded the abstract semantics uses $e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}$ and $(\textbf{success}\ \widehat{vs}; \widehat{\sigma}') \in \widehat{Res}$, as compared to $e; \sigma \xRightarrow[\text{expr}]{} \textbf{success}\ v; \sigma'$ in the concrete semantics.

- Typing is now done using an abstract judgment $\widehat{vs} \,\widehat{:}\, t$ which works analogously to the concrete one except t is over-approximates the concrete type of values, because the given value shape $\widehat{vs}$ is itself over-approximated. This means that the corresponding abstract subtyping $t \mathrel{\widehat{<:}} t'$ relation must take into account the over-approximation of t and thus that it may be possible that both $t \mathrel{\widehat{<:}} t'$ and $t \mathrel{\widehat{\not<:}} t'$ is true.

- To check whether a particular constructor is possible, we use the auxiliary function $\widehat{\text{unfold}}(\widehat{vs}, t)$ which produces a refined value of type t if possible—splitting alternative constructors in case of refinements—and otherwise produces **error** if the value is possibly not an element of t.

- We generalize the $\widehat{\text{ResultSet}}$ domain to work with sequences of values on successful evaluation for representing results of expression sequences $\underline{e}$. This is straightforward since a sequence of expressions must produce a sequence of values of the same size, and a fixed product of lattice domains forms a lattice (essentially the iterated Cartesian product).

Expressions (General)

$$\textsc{AE-Val}\ \dfrac{}{n;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \textbf{success}\ n;\widehat{\sigma}}$$

$$\textsc{AE-Bin}\ \dfrac{\{\!|e_1 \otimes e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-bin}]{} \widehat{Res}|\!\}}{e_1 \otimes e_2;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\textsc{AE-Asgn}\ \dfrac{\{\!|x = e;\widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} \widehat{Res}|\!\}}{x = e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\textsc{AE-Seq}\ \dfrac{\{\!|x = e;\widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} \widehat{Res}|\!\}}{x = e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\textsc{AE-Cons}\ \dfrac{\{\!|k(\underline{e});\widehat{\sigma} \xRightarrow[\text{a-expr-cons}]{} \widehat{Res}|\!\}}{k(\underline{e});\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\textsc{AE-Set}\ \dfrac{\{\!|\{\underline{e}\};\widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} \widehat{Res}|\!\}}{\{\underline{e}\};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

$$\textsc{AE-Fail}\ \dfrac{}{\textbf{fail};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \textbf{fail};\widehat{\sigma}}$$

$$\textsc{AE-If}\ \dfrac{\{\!|\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-if}]{} \widehat{Res}|\!\}}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res}}$$

Expression Sequences

$$\textsc{AES}\ \dfrac{\{\!|\underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \widehat{Res}|\!\}}{\underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res}}$$

$$\textsc{AES-Emp}\ \dfrac{}{\varepsilon;\widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \textbf{success}\ \varepsilon;\widehat{\sigma}}$$

$$\textsc{AES-More}\ \dfrac{\begin{array}{cc} e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} & (\textbf{success}\ \widehat{vs};\widehat{\sigma}'') \in \widehat{Res} \\ \underline{e'};\widehat{\sigma}'' \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star}' & (\textbf{success}\ \underline{\widehat{vs}'};\widehat{\sigma}') \in \widehat{Res\star}' \end{array}}{e,\underline{e'};\widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \textbf{success}\ \widehat{vs},\underline{\widehat{vs}'};\widehat{\sigma}'}$$

$$\textsc{AES-Exc1}\ \dfrac{e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res}}{e,\underline{e'};\widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \widehat{exres};\widehat{\sigma}'}$$

$$\textsc{AES-Exc2}\ \dfrac{\begin{array}{cc} e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} & (\textbf{success}\ \widehat{vs};\widehat{\sigma}'') \in \widehat{Res} \\ \underline{e'};\widehat{\sigma}'' \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star}' & (\widehat{exres};\widehat{\sigma}') \in \widehat{Res\star}' \end{array}}{e,\underline{e'};\widehat{\sigma} \xRightarrow[\text{a-expr}\star\text{-1}]{} \widehat{exres};\widehat{\sigma}'}$$

Figure 7.2: Abstract Operational Semantics Rules for Basic Expressions

Binary Expression

$$\textsc{AE-Bin-Sucs}\ \dfrac{\begin{array}{c} e_1;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\textbf{success}\ \widehat{v}_1;\widehat{\sigma}'') \in \widehat{Res} \\ e_2;\widehat{\sigma}'' \xRightarrow[\text{a-expr}]{} \widehat{Res}' \qquad (\textbf{success}\ \widehat{v}_2;\widehat{\sigma}') \in \widehat{Res}'' \\ \widehat{vs}_1 \otimes \widehat{vs}_2 = \widehat{VRes} \qquad \widehat{Res}' = \bigsqcup \left\{ \widehat{vres};\widehat{\sigma}' \;\middle|\; \widehat{vres} \in \widehat{VRes} \right\} \end{array}}{e_1 \otimes e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-bin}]{} \widehat{Res}'}$$

$$\textsc{AE-Bin-Exc1}\ \dfrac{e_1;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res}}{e_1 \otimes e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-bin}]{} \widehat{exres};\widehat{\sigma}'}$$

$$\textsc{AE-Bin-Sucs}\ \dfrac{\begin{array}{c} e_1;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\textbf{success}\ \widehat{v}_1;\widehat{\sigma}'') \in \widehat{Res} \\ e_2;\widehat{\sigma}'' \xRightarrow[\text{a-expr}]{} \widehat{Res}' \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res}' \end{array}}{e_1 \otimes e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-bin}]{} \widehat{exres};\widehat{\sigma}'}$$

Assignment Expression

$$\textsc{AE-Asgn-Sucs}\ \dfrac{\begin{array}{c} \textbf{local}\ t\ x \vee \textbf{global}\ t\ x \qquad e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \\ (\textbf{success}\ \widehat{vs};\widehat{\sigma}') \in \widehat{Res} \qquad \widehat{vs} \mathrel{\widehat{:}} t' \qquad t' \mathrel{\widehat{<:}} t \end{array}}{x = e;\widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} \textbf{success}\ \widehat{vs};\widehat{\sigma}'[x \mapsto (\text{ff}, \widehat{vs})]}$$

$$\textsc{AE-Asgn-Err}\ \dfrac{\begin{array}{c} \textbf{local}\ t\ x \vee \textbf{global}\ t\ x \qquad e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \\ (\textbf{success}\ \widehat{vs};\widehat{\sigma}') \in \widehat{Res} \qquad \widehat{vs} \mathrel{\widehat{:}} t' \qquad t' \mathrel{\widehat{\not<:}} t \end{array}}{x = e;\widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} \textbf{error};\widehat{\sigma}'}$$

$$\textsc{AE-Asgn-Exc}\ \dfrac{e;\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res}}{x = e;\widehat{\sigma} \xRightarrow[\text{a-expr-asgn}]{} \widehat{exres};\widehat{\sigma}'}$$

Sequencing Expression

$$\textsc{AE-Seq-Sucs}\ \dfrac{e_1, e_2;\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\textbf{success}\ \widehat{vs}_1, \widehat{vs}_2;\widehat{\sigma}') \in \widehat{Res\star}}{e_1; e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} \textbf{success}\ \widehat{vs}_2;\widehat{\sigma}'}$$

$$\textsc{AE-Seq-Exc}\ \dfrac{e_1, e_2;\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res\star}}{e_1; e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-seq}]{} \widehat{exres};\widehat{\sigma}'}$$

Figure 7.3: Abstract Operational Semantics Rules for Basic Expressions (Cont.)

Constructor Expression

$$\textsc{AE-Cons-Sucs}\ \dfrac{\begin{array}{c}\textbf{data}\ at = \ldots|k(\underline{t})|\ldots \qquad \underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \\ (\textbf{success}\ \underline{\widehat{vs}};\widehat{\sigma}') \in \widehat{Res\star} \qquad \underline{\widehat{vs}\ \widehat{:}\ t'} \qquad \underline{t'\ \widehat{<:}\ t}\end{array}}{k(\underline{e});\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \textbf{success}\ k(\underline{\widehat{vs}});\widehat{\sigma}'}$$

$$\textsc{AE-Cons-Err}\ \dfrac{\begin{array}{c}\textbf{data}\ at = \ldots|k(\underline{t})|\ldots \qquad \underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \\ (\textbf{success}\ \underline{\widehat{vs}};\widehat{\sigma}') \in \widehat{Res\star} \qquad \underline{\widehat{vs}\ \widehat{:}\ t'} \qquad \exists i.t'_i\ \widehat{\not<:}\ t_i\end{array}}{k(\underline{e});\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \textbf{error};\widehat{\sigma}'}$$

$$\textsc{AE-Cons-Exc}\ \dfrac{\underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res\star}}{k(\underline{e});\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{exres};\widehat{\sigma}'}$$

Set Expression

$$\textsc{AE-Set-Sucs}\ \dfrac{\underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\textbf{success}\ \underline{vs};\widehat{\sigma}') \in \widehat{Res\star}}{\{\underline{e}\};\widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} \textbf{success}\ \{\widehat{\underline{vs}}\};\widehat{\sigma}'}$$

$$\textsc{AE-Set-Exc}\ \dfrac{\underline{e};\widehat{\sigma} \xRightarrow[\text{a-expr}\star]{} \widehat{Res\star} \qquad (\underline{exres};\widehat{\sigma}') \in \widehat{Res\star}}{\{\underline{e}\};\widehat{\sigma} \xRightarrow[\text{a-expr-set}]{} \underline{exres};\widehat{\sigma}'}$$

If Expression

$$\textsc{AE-If-T}\ \dfrac{\begin{array}{c}e_{\text{cond}};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\textbf{success}\ \widehat{vs};\widehat{\sigma}'') \in \widehat{Res} \\ \textbf{success}\ \text{true}() \in \widehat{\text{unfold}}(\widehat{vs}, \text{Bool}) \qquad e_1;\widehat{\sigma}'' \xRightarrow[\text{a-expr}]{} \widehat{Res}'\end{array}}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-if}]{} \widehat{Res}'}$$

$$\textsc{AE-If-F}\ \dfrac{\begin{array}{c}e_{\text{cond}};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\textbf{success}\ \widehat{vs};\widehat{\sigma}'') \in \widehat{Res} \\ \textbf{success}\ \text{false}() \in \widehat{\text{unfold}}(\widehat{vs}, \text{Bool}) \qquad e_2;\widehat{\sigma}'' \xRightarrow[\text{a-expr}]{} \widehat{Res}'\end{array}}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-if}]{} \widehat{Res}'}$$

$$\textsc{AE-If-Er}\ \dfrac{\begin{array}{c}e_{\text{cond}};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\textbf{success}\ \widehat{vs};\widehat{\sigma}') \in \widehat{Res} \\ \textbf{error} \in \widehat{\text{unfold}}(\widehat{vs}, \text{Bool})\end{array}}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-if}]{} \textbf{error};\widehat{\sigma}'}$$

$$\textsc{AE-IF-Ex}\ \dfrac{e_{\text{cond}};\widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\widehat{exres};\widehat{\sigma}') \in \widehat{Res}}{\textbf{if}\ e_{\text{cond}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2;\widehat{\sigma} \xRightarrow[\text{a-expr-if}]{} \widehat{exres};\widehat{\sigma}'}$$

Figure 7.4: Abstract Operational Semantics Rules for Basic Expressions (Cont. 2)

Abstract Semantic Operators

We have already seen how the rules for basic expressions have been systematically translated, including the required checks. As discussed the same is also required for semantic operations, where we in particular will discuss operators (including unary and binary), and pattern matching.

Semantic Operators In general for each semantic operator $[\![\oplus]\!]$ corresponding to a language operator $\oplus$, we need to have a corresponding abstract semantic operator $[\![\widehat{\oplus}]\!]$ which works with the abstracted values of the target type, e.g. for $[\![+]\!] \in \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ we need $[\![\widehat{+}]\!] \in \widehat{\text{Interval}} \to \widehat{\text{Interval}} \to \widehat{\text{Interval}}$ and for $[\![\cup]\!] \in \wp(\text{Value}) \to \wp(\text{Value}) \to \wp(\text{Value})$ we need $[\![\widehat{\cup}]\!] \in \widehat{\text{SetShape}}(\widehat{\text{ValueShape}}) \to \widehat{\text{SetShape}}(\widehat{\text{ValueShape}}) \to \widehat{\text{SetShape}}(\widehat{\text{ValueShape}})$. Due to the overapproximative semantics, operands might get evaluated to values that are less precise than values in the supported range for the target abstract semantics operator, and in that case we need to consider both cases of where the abstract value lies in target range and where it lies outside of target range. We can do this by introducing rules that check these potential need for coercion when evaluating the syntactic operators, e.g. the following rules for binary operators:

$$\text{BOP}\,\frac{\{\!|\widehat{vs}_1 \oplus \widehat{vs}_2 =_1 \widehat{VRes}|\!\}}{\widehat{vs}_1 \oplus \widehat{vs}_2 = \widehat{VRes}}$$

$$\text{BOP1-1}\,\frac{\begin{array}{c}[\![\widehat{\oplus}]\!] \in [\![\widehat{t_1}]\!] \times [\![\widehat{t_2}]\!] \to [\![\widehat{t'}]\!] \\ \textbf{value}\ \widehat{vs_1}' \in \widehat{\text{unfold}}(\widehat{vs_1}, t_1) \qquad \textbf{value}\ \widehat{vs_2}' \in \widehat{\text{unfold}}(\widehat{vs_2}, t_2)\end{array}}{\widehat{vs}_1 \oplus \widehat{vs}_2 =_1 \textbf{value}\ [\![\widehat{\oplus}]\!](\widehat{vs_1}', \widehat{vs_2}')}$$

$$\text{BOP1-2}\,\frac{\begin{array}{c}[\![\widehat{\oplus}]\!] \in [\![\widehat{t_1}]\!] \times [\![\widehat{t_2}]\!] \to [\![\widehat{t'}]\!] \\ \textbf{error} \in \widehat{\text{unfold}}(\widehat{vs_1}, t_1) \vee \textbf{error} \in \widehat{\text{unfold}}(\widehat{vs_2}, t_2)\end{array}}{\widehat{vs}_1 \oplus \widehat{vs}_2 =_1 \textbf{error}}$$

Pattern matching The concrete pattern matching judgments have forms $\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho}$ for a concrete pattern p matched against value v under store σ—used for matching target value against assigned variable values—producing a sequence of binding environments (each $\rho \in \text{BindingEnv} = \text{Var} \rightharpoonup \text{Value}$), and $\sigma \vdash \star p \overset{?}{:=} v \mid \mathbb{V} \xRightarrow[\text{match}\star]{} \underline{\rho}$ for a sequence pattern $\star p$ with similar construction but also having a set of already matched values $\mathbb{V}$ to track already the set of already matched partitions of values for non-deterministic matches of collection elements.

The corresponding abstract matching judgment for ordinary patterns has form $\widehat{\sigma} \vdash p \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}$ where we pattern match on value shapes $\widehat{vs}$ and produce an output $\widehat{\varrho_R}$ representing the set of possible match results. Each match result triple $(\widehat{\sigma}', \widehat{vs}', \widehat{\varrho}) \in \widehat{\varrho_R}$, has an abstract store $\widehat{\sigma}'$ and abstract value shape $\widehat{vs}'$ that soundly refine the input store $\widehat{\sigma}$ and value $\widehat{vs}$—so $\widehat{\sigma}' \sqsubseteq \widehat{\sigma}$ and $\widehat{vs}' \sqsubseteq \widehat{vs}$—assuming that matching the pattern p produced the abstract set of binding environments $\widehat{\varrho}$; this refinement is important to maintain precision during success and failure of individual pattern matches. More formally, we can define refined binding environments as follows:

$$\widehat{\varrho_R} \in \widehat{\text{RBindingEnvs}} = \wp\left(\widehat{\text{Store}} \times \widehat{\text{ValueShape}} \times \wp\left(\widehat{\text{BindingEnv}}\right)\right)$$
$$\widehat{\rho} \in \widehat{\text{BindingEnv}} = \text{Var} \rightharpoonup \widehat{\text{ValueShape}}$$

The AP-Var rule collects three possible rules for abstract pattern matching against variable patterns: AP-Var-Uni, AP-Var-Fail and AP-Var-Bind. AP-Var-Uni pattern matches the values shape against the possibly assigned value shape in the store, refining both shapes assuming that they are equal. The AP-Var-Fail rule works analogously, but captures the case where they are possibly not equal. AP-Var-Bind captures the case where the variable is unassigned in the store, and thus returns a binding of the variable to the input value shape.

$$\textsc{AP-Var}\;\frac{\{\!\!|\widehat{\sigma} \vdash x \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match-v1}]{} \widehat{\varrho_R}|\!\!\}}{\widehat{\sigma} \vdash x \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}}$$

$$\textsc{AP-Var-Uni}\;\frac{\widehat{\sigma}(x) = (b, \widehat{vs}') \qquad \widehat{vs}' \neq \bot_{\widehat{\text{VS}}} \qquad \widehat{vs}'' \in (\widehat{vs} \mathrel{\widehat{=}} \widehat{vs'})}{\widehat{\sigma} \vdash x \overset{?}{:=} \widehat{vs} \mid \widehat{vs}' \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}[x \mapsto (\text{ff}, \widehat{vs}'')], \widehat{vs}'', \{[]\})}$$

$$\textsc{AP-Var-Fail}\;\frac{\widehat{\sigma}(x) = (b, \widehat{vs}') \qquad \widehat{vs}' \neq \bot_{\widehat{\text{VS}}} \qquad (\widehat{vs}'', \widehat{vs}''') \in (\widehat{vs} \mathrel{\widehat{\neq}} \widehat{vs'})}{\widehat{\sigma} \vdash x \overset{?}{:=} \widehat{vs} \mid \widehat{vs}' \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}[x \mapsto (\text{ff}, \widehat{vs}''')], \widehat{vs}'', \emptyset)}$$

$$\textsc{AP-Var-Bind}\ \frac{\widehat{\sigma}(x) = (\mathrm{tt}, \widehat{vs}')}{\widehat{\sigma} \vdash x \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match-v}]{} (\widehat{\sigma}[x \mapsto (\mathrm{tt}, \bot_{\widehat{\mathrm{VS}}})], \widehat{vs}, \{[x \mapsto \widehat{vs}]\})}$$

The rule AP-Cons collects two rules for pattern matching against constructors: AP-Cons-Sucs and AP-Cons-Fail. AP-Cons-Sucs captures the case where the target value shape can be unfolded to the constructor required by the pattern, and then continues to match the inner patterns and merge their results. AP-Cons-Fail in contrast considers the case where target value can be unfolded to some other constructor or is not a value of target type (producing **error**); in this case, the set of binding environments is empty, and the input value is refined to exclude the target pattern constructor from the result value shape if possible.

$$\textsc{AP-Cons}\ \frac{\{\!|\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{:=} \widehat{vs} \mid at \xRightarrow[\text{a-match-cons}]{} \widehat{\varrho_R}|\!\}}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}}$$

$$\textsc{AP-Cons-Sucs}\ \frac{\begin{array}{c}\textbf{data}\ at = \cdots \mid k(\underline{t}) \mid \ldots \qquad (\textbf{value}\ k(\underline{\widehat{vs}'})) \in \widehat{\mathrm{unfold}}(\widehat{vs}, at) \\ \widehat{\sigma} \vdash p_1 \overset{?}{:=} \widehat{vs}'_1 \xRightarrow[\text{a-match}]{} \widehat{\varrho_{R1}} \ \ldots\ \widehat{\sigma} \vdash p_{\mathrm{n}} \overset{?}{:=} \widehat{vs}'_{\mathrm{n}} \xRightarrow[\text{a-match}]{} \widehat{\varrho_{R\mathrm{n}}} \\ (\widehat{\sigma}'_1, \widehat{vs}'_1, \widehat{\varrho_1}) \in \widehat{\varrho_{R1}} \ \ldots\ (\widehat{\sigma}'_{\mathrm{n}}, \widehat{vs}'_{\mathrm{n}}, \widehat{\varrho_{\mathrm{n}}}) \in \widehat{\varrho_{R\mathrm{n}}}\end{array}}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{:=} \widehat{vs} \mid at \xRightarrow[\text{a-match-cons}]{} (\bigsqcap_i \widehat{\sigma}_i, k(\underline{\widehat{vs}'}), \widehat{\mathrm{merge}}(\underline{\widehat{\varrho}}))}$$

$$\textsc{AP-Cons-Fail}\ \frac{\begin{array}{c}\textbf{data}\ at = \cdots \mid k(\underline{t}) \mid \ldots \\ (\textbf{value}\ k'(\underline{\widehat{vs}'})) \in \widehat{\mathrm{unfold}}(\widehat{vs}, at) \land k' \neq k \lor \textbf{error} \in \widehat{\mathrm{unfold}}(\widehat{vs}, at)\end{array}}{\widehat{\sigma} \vdash k(\underline{p}) \overset{?}{:=} \widehat{vs} \mid at \xRightarrow[\text{a-match-cons}]{} (\widehat{\sigma}, \widehat{\mathrm{exclude}}(\widehat{vs}, k), \emptyset)}$$

The AP-Set rule collects two rules: AP-Set-Sucs and AP-Set-Fail. AP-Set-Sucs tries to unfold the target value shape as a set, and then proceeds by pattern matching the against the inner sequence of patterns, using the content shape and cardinality of the set. AP-Set-Fail produces a failure result with an empty binding environment set, if the target value shape is possibly not a set.

$$\textsc{AP-Set}\ \frac{\{\!|\widehat{\sigma} \vdash \{\underline{\star p}\} \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match-set}]{} \widehat{\varrho_R}|\!\}}{\widehat{\sigma} \vdash \{\underline{\star p}\} \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}}$$

$$\textsc{AP-Set-Sucs}\ \frac{\textbf{value}\ \{\widehat{vs}'\}_{[l;u]} \in \widehat{\text{unfold}}(\widehat{vs}, \textbf{set}\langle\textbf{value}\rangle) \qquad \widehat{\sigma} \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}}{\widehat{\sigma} \vdash \{\underline{\star p}\} \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match-set}]{} \widehat{\varrho_R}}$$

$$\textsc{AP-Set-Fail}\ \frac{\textbf{error} \in \widehat{\text{unfold}}(\widehat{vs}, \textbf{set}\langle\textbf{value}\rangle)}{\widehat{\sigma} \vdash \{\underline{\star p}\} \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match-set}]{} (\widehat{\sigma}, \widehat{vs}, \emptyset)}$$

The rule APL collects nine abstract rules for matching sequences of patterns, where each rule corresponding to a concrete pattern matching rule for sequence of patterns. Notice that our abstract rules do not backtrack like our concrete rules since we only have a single fixed shape abstraction for all the elements in a collection.

$$\text{APL}\ \frac{\{\!|\widehat{\sigma} \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} \widehat{\varrho_R}|\!\}}{\widehat{\sigma} \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}}$$

There are two possible abstract rules for pattern matching against an empty sequence of patterns: APL-Emp-Both and APL-Emp-Pat. APL-Emp-Both accounts for the case where the abstracted sequence is possibly empty (has cardinality 0) producing a set containing an empty binding environment (indicating a successful match with no variable bindings). APL-Emp-Pat conversely accounts for the case where abstracted sequence is possibly non-empty, and therefore produces an empty set of binding environments (indicating a failing match).

$$\textsc{APL-Emp-Both}\ \frac{l \leq u \qquad l = 0}{\widehat{\sigma} \vdash \varepsilon \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} (\widehat{\sigma}, \{\bot_{\widehat{\text{VS}}}\}_0, \{[]\})}$$

$$\textsc{APL-Emp-Pat}\ \frac{l \leq u \qquad u \neq 0}{\widehat{\sigma} \vdash \varepsilon \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} (\widehat{\sigma}, \{\widehat{vs}\}_{[\max(1,l),u]}, \emptyset)}$$

There is similarly two rules for pattern matching against a non-empty pattern sequence starting, where the first element is an ordinary pattern: APL-Emp-Val and APL-More-Pat. APL-Emp-Val accounts for the case where the abstracted set of elements is possibly empty, and so produces an empty set of bindings environments. APL-More-Pat accounts for the case where the abstracted set of elements is non-empty, and therefore i) pattern matches the shape of elements against the first pattern,

ii) matches the rest of the pattern sequence against the element shape where the cardinality has been decreased by one, and iii) merges the resulting binding set.

$$\textsc{APL-Emp-Val}\;\frac{l \le u \qquad l = 0}{\widehat{\sigma} \vdash p, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} (\widehat{\sigma}, \{\widehat{vs}\}_0, \emptyset)}$$

$$\textsc{APL-More-Pat}\;\frac{\begin{array}{c} l \le u \qquad u \ne 0 \qquad \widehat{\sigma} \vdash p \overset{?}{:=} \widehat{vs} \xRightarrow[\text{a-match}]{} \widehat{\varrho_R}' \\ \widehat{\sigma} \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l-1;u-1] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}'' \\ (\widehat{\sigma}', \widehat{vs}', \widehat{\varrho}') \in \widehat{\varrho_R}' \qquad (\widehat{\sigma}'', \widehat{vs}'', \widehat{\varrho}'') \in \widehat{\varrho_R}'' \\ \widehat{\varrho_R}''' = \{(\widehat{\sigma}' \sqcup \widehat{\sigma}'', \{\widehat{vs}'\}_1 \sqcup \widehat{vs}'', \widehat{\text{merge}}(\widehat{\varrho}', \widehat{\varrho}''))\} \end{array}}{\widehat{\sigma} \vdash p, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} \widehat{\varrho_R}'''}$$

Finally, there are five rules for matching against star patterns. Three of the rules are applicable when a value is bound in the store: APL-Star-Uni, APL-Star-Pat-Fail and APL-Star-Val-Fail. APL-Star-Uni is the most elaborate rule, since it has to account for the case where a subset matched the assigned value in the store; essentially, it follows these steps:

1. It checks whether the assigned value to the variable x can possibly unfold to a set value.

2. It calculates the overlapping shape and cardinality, if there is any overlap.

3. It matches the rest of the pattern sequence with a refined store, and interval abstract the cardinality of the rest of elements (essentially uses the interval difference).

4. It additionally takes into account the case where the concrete sets do not match even though their value shapes and cardinality were compatible and thus adds an empty set of binding environments as a possibility. For example, the assigned value shape and the analyzed value shape could both be $\{[1;2]\}_1$ which would have both $\{1\}$ and $\{2\}$ as valid concretizations, which would not match in a concrete evaluation.

The APL-Star-Pat-Fail rule accounts for the case where either the assigned an analyzed shapes are incompatible, or where the cardinalities are incompatible, producing an empty set of binding environments. Finally, APL-Star-Val-Fail accounts for the case where the assigned value shape is not a set at all.

$$\textsc{APL-Star-Uni}\;\frac{\begin{array}{c} l \leq u \qquad\qquad \widehat{\sigma}(x) = (b, \widehat{vs}') \\ \mathbf{value}\ \{\widehat{vs}''\}_{[l'';u'']} \in \widehat{\text{unfold}}(\widehat{vs}, \mathbf{set}\langle\mathbf{value}\rangle) \\ \widehat{vs}''' \in (\widehat{vs} \mathbin{\widehat{=}} \widehat{vs}'') \qquad\qquad [l';u'] = [l'';u''] \sqcap [l;u] \\ [l';u'] \neq \bot \qquad\qquad [l''';u'''] = [\min(0, l - u'); u - l'] \\ \widehat{\sigma}[x \mapsto (\text{ff}, \{\widehat{vs}''\}_{[l';u']})] \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l''';u'''] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}'' \\ \widehat{\varrho_R}''' = \left\{ (\widehat{\sigma}'', \{\widehat{vs}''' \sqcup \widehat{vs}''''\}_{[l;u]}, \widehat{\varrho}'') \;\middle|\; (\widehat{\sigma}'', \{\widehat{vs''''}\}_{[l'''';u'''']}, \widehat{\varrho}'') \in \widehat{\varrho}''_R \right\} \\ \widehat{\varrho_R}' = \widehat{\varrho_R}''' \cup \{(\widehat{\sigma}[x \mapsto (\text{ff}, \{\widehat{vs}''\}_I)], \{\widehat{vs}\}_{[l;u]}, \emptyset)\} \end{array}}{\widehat{\sigma} \vdash \star x, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star\text{-1}]{} \widehat{\varrho_R}'}$$

$$\textsc{APL-Star-Pat-Fail}\;\frac{\begin{array}{c} l \leq u \qquad\qquad \widehat{\sigma}(x) = (b, \widehat{vs}') \\ \mathbf{value}\ \{\widehat{vs}''\}_{[l'';u'']} \in \widehat{\text{unfold}}(\widehat{vs}, \mathbf{set}(\mathbf{value})) \\ (\widehat{vs} \mathbin{\widehat{\neq}} \widehat{vs}'') \neq \emptyset \vee [l'';u''] \sqcap [l;u] = \bot \\ \widehat{\varrho_R}' = \{(\widehat{\sigma}[x \mapsto (\text{ff}, \{\widehat{vs}''\}_I)], \{\widehat{vs}\}_{[l;u]}, \emptyset)\} \end{array}}{\widehat{\sigma} \vdash \star x, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}'}$$

$$\textsc{APL-Star-Val-Fail}\;\frac{l \leq u \qquad \widehat{\sigma}(x) = (b, \widehat{vs}') \qquad \mathbf{error} \in \widehat{\text{unfold}}(\widehat{vs}, \mathbf{set}(\mathbf{value}))}{\widehat{\sigma} \vdash \star x, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l;u] \xRightarrow[\text{a-match}\star]{} (\widehat{\sigma}[x \mapsto (\text{ff}, \widehat{vs}')], \{\widehat{vs}\}_{[l;u]}, \emptyset)}$$

The two final rules for star patterns—APL-Star-Re and APL-Star-Exh—concern when the target variable is possibly not assigned in the store. APL-Star-Re tries to bind a subcollection of elements and continue matching the rest of the sequence, where APL-Star-Exh abstracts over the case where backtracking has exhausted all possible partitions and thus produces no binding environment.

$$\textsc{APL-Star-Re}\;\frac{\begin{array}{c} l \leq u \qquad\qquad \widehat{\sigma}(x) = (\mathrm{tt}, \widehat{vs}') \\ \widehat{\sigma}[x \mapsto (\mathrm{tt}, \bot_{\widehat{\mathrm{VS}}})] \vdash \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [0; u] \xRightarrow[\text{a-match}\star]{} \widehat{\varrho_R}'' \\ (\widehat{\sigma}'', \{\widehat{vs}''\}_{[l'';u'']}, \widehat{\varrho}'') \in \widehat{\varrho_R}'' \\ \widehat{\varrho}'_R = (\widehat{\sigma}'', \{\widehat{vs}\}_{[l;u]}, \widehat{\mathrm{merge}}(\{[x \mapsto \{\widehat{vs}\}_{[l';u']}]\}, \widehat{\varrho}'')) \end{array}}{\widehat{\sigma} \vdash \star x, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l; u] \xRightarrow[\text{a-match}\star\text{-1}]{} \widehat{\varrho}'_R}$$

$$\textsc{APL-Star-Exh}\;\frac{l \leq u \qquad \widehat{\sigma}(x) = (\mathrm{tt}, \widehat{vs}')}{\widehat{\sigma} \vdash \star x, \underline{\star p} \overset{?}{:=} \widehat{vs} \mid [l; u] \xRightarrow[\text{a-match}\star\text{-1}]{} (\widehat{\sigma}, \{\widehat{vs}\}_{[l;u]}, \varnothing)}$$

Trace Memoization and Traversals

In abstract interpretation and static program analysis in general, the main challenge is usually how to perform fixed-point calculation for unbounded loops and non-structural recursion. In Schmidt-style abstract interpretation, the main technique to handle this is *trace memoization* [Schmidt, 1998]. The core idea of *trace memoization* is to detect non-structural re-evaluation of the same program element—i.e., where the evaluation of a program element is recursively dependent on the evaluation of itself, like a **while**-loop or traversal—and in such case apply a widening on the input values and state in order to get an over-approximating finite circular dependency in the trace of the program element. This circular dependent trace can be seen as a continuous function [Schmidt, 1998; Rosendahl, 2013], and to get an output we perform classical fixed-point iteration.

The trace memoization strategies suggested by existing work, do however not directly apply to the constructs present in Rascal. Schmidt [1998] presents a simple technique which only works for constructs like **while**-loops where the self-recursive evaluation happens in a tail position and so the resulting output is the same as the resulting output as recursive dependency. The technique suggested by Rosendahl [2013] supports more general recursive dependencies, but only works for finite input domains, where our input domains are all infinite.

We have therefore further extended the trace memoization strategy suggested by Rosendahl [2013] to work with abstract inputs that are from infinite domains. The extension is still terminating and sound—so the function representing the potentially circular trace is continuous—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. In particular, our technique works as follows:

- Assume all our domains are widening lattices. Consider a each non-structurally 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 $\widehat{M} \in \widehat{\text{PInput}} \rightarrow \widehat{\text{Input}} \times \widehat{\text{Output}}$ 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 for example use the corresponding type from the domain Type as input to the memoization map.[4] So for values 1 and $[2; 3]$ we would use **int**, while for fieldaccessexpr(Expr, **str**) 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 empty, 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 is then three 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 $i \sqsubseteq_{\widehat{\text{Input}}} i'$. In this case we return o' directly as result without further evaluation; this correctly preserves continuity of the complete semantics since we are returning a less precise result, the one for the less precise input i', than the one that is expected for the current input i.

  Widen The corresponding input partition key is in the memoization map, but an unrelated or more precise input is stored, i.e.,

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

$\widehat{M}(\alpha_{\widehat{PI}}(i)) = (i'', o'')$ 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 $\widehat{M}' = [\alpha_{\widehat{PI}}(i) \mapsto (i', o_{\text{prev}})]$, where o_{prev} is the output of the last fixed-point iteration for i'. If the output of the evaluation of i', o' is different than the previous output o_{prev}, we continue fixed-point iteration.

Miss The corresponding input partition key is not in the memoization map, so $\alpha_{\widehat{PI}}(i) \notin \text{dom } \widehat{M}$ and so we continue evaluation of i with an updated map $\widehat{M}' = \widehat{M}[\alpha_{\widehat{PI}}(i) \mapsto (i, o_{\text{prev}})]$ passed down recursively that assigns the input i and the output of the previous iteration for i in the fixed-point calculation o_{prev} (initially $\bot_{\widehat{\text{Output}}}$) to the relevant input partition key. If the resulting output o is equivalent to the previous output o_{prev} then we stop the fixed-point calculation, and otherwise we perform a new iteration with an updated previous output value $o'_{\text{prev}} = o_{\text{prev}} \nabla o$.

We will now show how we constructor our abstract operational semantics rules for visitors. The required steps to get the abstract rules for non-structurally recursive constructs are the same as the ones for translating rules of basic constructs, but we must additionally take into account the memoization map and the fixed-point calculation.

The first step is to construct the top-level abstract evaluation rules for evaluating the visit expressions, each translated from a concrete evaluation rule for the visit expression. This is done as usual using the technique presented in the section. The AE-Visit rule collects the four possible cases: AE-Visit-Sucs, AE-Visit-Fail, AE-Visit-Exc1 and AE-Visit-Exc2.

$$\textsc{AE-Visit}\ \frac{\{\!|\,\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xrightarrow[\text{a-expr-visit}]{} \widehat{Res}\,|\!\}}{\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xrightarrow[\text{a-expr}]{} \widehat{Res}}$$

AE-Visit-Sucs handles the case where both the evaluation of the target expression and the traversal succeeds. AE-Visit-Fail handles the case where the traversal fails, and produces a success value with the refined value shape given by **fail**; using the refined value is important to maintain precision, since using the initial input to the traversal like the concrete evaluation would not take into account information learned by pattern matching. Finally, AE-Visit-Exc1 and AE-Visit-Exc2 handle the exceptional cases during evaluation of the target expression and the traversal, respectively.

$$\textsc{AE-Visit-Sucs}\ \frac{\begin{array}{cc} e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} & (\textbf{success}\ \widehat{vs}; \widehat{\sigma}'') \in \widehat{Res} \\ \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' & (\textbf{success}\ \widehat{vs}'; \widehat{\sigma}') \in \widehat{Res}' \end{array}}{\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} \textbf{success}\ \widehat{vs}'; \widehat{\sigma}'}$$

$$\textsc{AE-Visit-Fail}\ \frac{\begin{array}{cc} e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} & (\textbf{success}\ \widehat{vs}; \widehat{\sigma}'') \in \widehat{Res} \\ \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' & (\textbf{fail}\ \widehat{vs}'; \widehat{\sigma}') \in \widehat{Res}' \end{array}}{\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} \textbf{success}\ \widehat{vs}'; \widehat{\sigma}'}$$

$$\textsc{AE-Visit-Exc1}\ \frac{e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} \qquad (\widehat{exres}; \widehat{\sigma}') \in \widehat{Res}}{\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} \widehat{exres}; \widehat{\sigma}'}$$

$$\textsc{AE-Visit-Exc2}\ \frac{\begin{array}{cc} e; \widehat{\sigma} \xRightarrow[\text{a-expr}]{} \widehat{Res} & (\textbf{success}\ \widehat{vs}; \widehat{\sigma}'') \in \widehat{Res} \\ \underline{cs}; \widehat{vs}; \widehat{\sigma}'' \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}' & (\textbf{error}; \widehat{\sigma}') \in \widehat{Res}' \end{array}}{\textbf{bottom-up visit}\ e\ \underline{cs}; \widehat{\sigma} \xRightarrow[\text{a-expr-visit}]{} \textbf{error}; \widehat{\sigma}'}$$

The next step is to translate the actual traversal rules for visit using the memoization technique. This requires setting up rules that follow the extended steps for trace memoization and passing around the memoization map between mutually recursive judgments. The AEBU rule initializes memoization process, by delegating to the memoized judgment with an initially empty memoization map.

$$\text{AEBU}\ \frac{[\,] \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}}{\underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit}]{} \widehat{Res}}$$

There are three rules for the memoized evaluation judgment—AEBUM-Hit, AEBUM-Widen and AEBUM-Miss—each corresponding to one of the described cases. The AEBUM-Hit finds a less precise input stored at the memoization partition key, and can therefore soundly use the memoized output result set. The AEBUM-Widen conversely finds a more precise input memoized at the partition key, and so must use the current input for widening to a less precise value shape; we then do a fixed-point calculation of the result of traversing the widened input with the bottom value $\bot_{\widehat{\text{VS}}}$ as initial pre-result. The AEBUM-Miss rule handles the case where the target value shape does not have a corresponding partition key in the memoization map, and so simply uses the current value for fixed-point calculation with $\bot_{\widehat{\text{VS}}}$ as initial pre-result.

$$\text{AEBUM-Hit}\ \frac{\widehat{M}(\alpha_{\text{Type}}(\widehat{vs})) = (\widehat{vs}', \widehat{Res}') \qquad \widehat{vs} \sqsubseteq_{\widehat{\text{RS}}} \widehat{vs}'}{\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}'}$$

$$\text{AEBUM-Widen}\ \frac{\begin{array}{c}\widehat{M}(\alpha_{\text{Type}}(\widehat{vs})) = (\widehat{vs}'', \widehat{Res}') \qquad \widehat{vs} \not\sqsubseteq_{\widehat{\text{RS}}} \widehat{vs}'' \qquad \widehat{vs}' = \widehat{vs} \nabla_{\widehat{\text{RS}}} (\widehat{vs} \sqcup \widehat{vs}'') \\ \widehat{M}; \bot_{\widehat{\text{RS}}} \vdash \underline{cs}; \widehat{vs}'; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-fix}]{} \widehat{Res}'\end{array}}{\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}'}$$

$$\text{AEBUM-Miss}\ \frac{\alpha_{\text{Type}}(\widehat{vs}) \notin \text{dom}\ \widehat{M} \qquad \widehat{M}; \bot_{\widehat{\text{RS}}} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-fix}]{} \widehat{Res}'}{\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}'}$$

The AEBUMF-Stop and AEBUMF-More rules are used for fixed-point calculation over the traversal result. The AEBUMF-Stop rule stops when evaluating the traversal body produces the same result as before iteration. Conversely, the AEBUMF-More takes an iteration where the result is strictly decreasing in precision, and so it needs to widen the output result states and repeat iteration. Because of monotonicity, it is not possible to get a result that gets more precise after a fixed-point iteration (otherwise, termination and the existence of fixed-point is not certain).

$$\text{AEBUMF-Stop}\ \frac{\begin{array}{c}\widehat{M}[\alpha_{\text{Type}}(\widehat{vs}) \mapsto (\widehat{vs}, \widehat{Res}_{\text{prev}})] \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-go}]{} \widehat{Res} \\ \widehat{Res}_{\text{prev}} = \widehat{Res}\end{array}}{\widehat{M}; \widehat{Res}_{\text{prev}} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-fix}]{} \widehat{Res}}$$

$$\text{AEBUMF-More}\ \frac{\begin{array}{c}\widehat{M}[\alpha_{\text{Type}}(\widehat{vs}) \mapsto (\widehat{vs}, \widehat{Res}_{\text{prev}})] \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-go}]{} \widehat{Res} \\ \widehat{Res}_{\text{prev}} \sqsubset \widehat{Res} \qquad\qquad \widehat{Res}'' = \widehat{Res}_{\text{prev}} \nabla_{\widehat{\text{RS}}} \widehat{Res} \\ \widehat{M}; \widehat{Res}'' \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-fix}]{} \widehat{Res}'\end{array}}{\widehat{M}; \widehat{Res}_{\text{prev}} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-fix}]{} \widehat{Res}'}$$

Having set-up the elaborate memoization machinery, we proceed by translating the traversal rules as usual. The only difference is that the memoization environment must be carried around and the memoized judgment must be used on recursion. The AEBUMG collect the four different bottom-up traversal rules each roughly corresponding to a single rule in the concrete semantics.[5]

$$\text{AEBUMG}\;\frac{\{\!|\widehat{M} \vdash \underline{cs};\widehat{vs};\widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \widehat{Res}|\!\}}{\widehat{M} \vdash \underline{cs};\widehat{vs};\widehat{\sigma} \xRightarrow[\text{a-bu-visit-memo-go}]{} \widehat{Res}}$$

The AEBUG-Sucs rule handles the case where the traversal succeeds both for the contained values [6] and the reconstructed top-level value. The AEBUG-Fail-Sucs rule handles the case where the traversal fails for the contained values and thus must be carried out only on the top-level value; here, we expect the reconstruction of values to always succeed, because the values returned in failure should be refinements of the original values. The AEBUG-Exc rule handles errors when traversing contained values, and AEBUG-Err rule handles the case where reconstruction produces an error.

$$\text{AEBUG-Sucs}\;\frac{\begin{array}{c}(\widehat{vs}', \underline{\widehat{vs}''}) \in \widehat{\text{children}}(\widehat{vs}) \qquad \widehat{M} \vdash \underline{cs};\underline{\widehat{vs}''};\widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \\ (\textbf{success}\ \underline{\widehat{vs}'''};\widehat{\sigma}') \in \widehat{Res\star} \qquad \widehat{\textbf{recons}}\ \widehat{vs}'\ \textbf{using}\ \underline{\widehat{vs}'''}\ \textbf{to}\ \widehat{RCRes} \\ (\textbf{success}\ \widehat{vs}'''') \in \widehat{RCRes} \qquad \widehat{M} \vdash \underline{cs};\widehat{vs}'''';\widehat{\sigma}' \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}'\end{array}}{\widehat{M} \vdash \underline{cs};\widehat{vs};\widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \widehat{Res}'}$$

$$\text{AEBUG-Fail-Sucs}\;\frac{\begin{array}{c}(\widehat{vs}', \underline{\widehat{vs}''}) \in \widehat{\text{children}}(\widehat{vs}) \qquad \widehat{M} \vdash \underline{cs};\underline{\widehat{vs}''};\widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \\ (\textbf{fail}\ \underline{\widehat{vs}'''};\widehat{\sigma}') \in \widehat{Res\star} \qquad \widehat{\textbf{recons}}\ \widehat{vs}'\ \textbf{using}\ \underline{\widehat{vs}'''}\ \textbf{to success}\ \widehat{vs}'''' \\ \widehat{M} \vdash \underline{cs};\widehat{vs}'''';\widehat{\sigma}' \xRightarrow[\text{a-bu-visit-memo}]{} \widehat{Res}'\end{array}}{\widehat{M} \vdash \underline{cs};\widehat{vs};\widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \widehat{Res}'}$$

$$\text{AEBUG-Exc}\;\frac{\begin{array}{c}(\widehat{vs}', \underline{\widehat{vs}''}) \in \widehat{\text{children}}(\widehat{vs}) \qquad \widehat{M} \vdash \underline{cs};\underline{\widehat{vs}''};\widehat{\sigma} \xRightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \\ (\textbf{error};\widehat{\sigma}') \in \widehat{Res\star}\end{array}}{\widehat{M} \vdash \underline{cs};\widehat{vs};\widehat{\sigma} \xRightarrow[\text{a-bu-visit-go}]{} \textbf{error};\widehat{\sigma}'}$$

[5]Not counting the additional rules for the $-$**break** version.

[6]We are here considering the case where the $\widehat{\text{children}}$ function was applied to data values and so produces a fixed sequence of heterogeneous value shapes. In case of collections or $\top$, it may produce a value shape-cardinality pair instead.

$$\textsc{AEBUG-Err}\ \frac{\begin{array}{c}(\widehat{vs}', \underline{\widehat{vs}''}) \in \widehat{\text{children}}(\widehat{vs}) \qquad \widehat{M} \vdash \underline{cs}; \underline{\widehat{vs}''}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star} \\ (\textbf{success}\ \underline{\widehat{vs}'''}; \widehat{\sigma}') \in \widehat{Res\star} \qquad \widehat{\textbf{recons}}\ \widehat{vs}'\ \textbf{using}\ \underline{\widehat{vs}'''}\ \textbf{to}\ \widehat{RCRes} \\ \textbf{error} \in \widehat{RCRes}\end{array}}{\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit-go}]{} \textbf{error}; \widehat{\sigma}'}$$

We do a similar translation with bottom-up traversal of a sequence of values. The AEBUS rule collects the four rules for traversing a sequence of expressions: AEBUSG-Emp, AEBUSG-More, AEBUSG-Exc1 and AEBUSG-Exc2. Note that in order we still need to carry around the memoization environment and recursively use the memoized judgment.

$$\textsc{AEBUS}\ \frac{\{\!|\widehat{M} \vdash \underline{cs}; \underline{\widehat{vs}}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star\text{-go}]{} \widehat{Res\star}|\!\}}{\widehat{M} \vdash \underline{cs}; \underline{\widehat{vs}}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star}}$$

The AEBUSG-EMP rule handles the case where the given abstract value shape is sequence, and so fails to match any case. The AEBUSG-More rule evaluates the first value shape in the sequence using the memoized bottom-up traversal rule, continues traversal of the rest of the sequences and combines the results. Finally, the AEBUSG-Exc1 and AEBUSG-Exc2 rules handle the cases where sub-traversals produced errors.

$$\textsc{AEBUSG-Emp}\ \frac{}{\widehat{M} \vdash \underline{cs}; \varepsilon; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star\text{-go}]{} \textbf{fail}\ \varepsilon; \widehat{\sigma}}$$

$$\textsc{AEBUSG-More}\ \frac{\begin{array}{c}\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit-memo}]{} \widehat{Res} \qquad (\widehat{vfres}; \widehat{\sigma}'') \in \widehat{Res} \\ \widehat{M} \vdash \underline{cs}; \underline{\widehat{vs}'}; \widehat{\sigma}'' \xrightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star}' \qquad (\widehat{vfres\star}'; \widehat{\sigma}') \in \widehat{Res\star}'\end{array}}{\widehat{M} \vdash \underline{cs}; \widehat{vs}, \underline{\widehat{vs}'}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star\text{-go}]{} \widehat{\text{vcombine}}(\widehat{vfres}, \widehat{vfres\star}'); \widehat{\sigma}'}$$

$$\textsc{AEBUSG-Exc1}\ \frac{\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit-memo}]{} \widehat{Res} \qquad (\textbf{error}; \widehat{\sigma}') \in \widehat{Res}}{\widehat{M} \vdash \underline{cs}; \widehat{vs}, \underline{\widehat{vs}'}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star\text{-go}]{} \textbf{error}; \widehat{\sigma}'}$$

$$\textsc{AEBUSG-Exc2}\ \frac{\begin{array}{c}\widehat{M} \vdash \underline{cs}; \widehat{vs}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit-memo}]{} \widehat{Res} \qquad (\widehat{vfres}; \widehat{\sigma}'') \in \widehat{Res} \\ \widehat{M} \vdash \underline{cs}; \underline{\widehat{vs}'}; \widehat{\sigma}'' \xrightarrow[\text{a-bu-visit}\star]{} \widehat{Res\star}' \qquad (\textbf{error}; \widehat{\sigma}') \in \widehat{Res\star}'\end{array}}{\widehat{M} \vdash \underline{cs}; \widehat{vs}, \underline{\widehat{vs}'}; \widehat{\sigma} \xrightarrow[\text{a-bu-visit}\star\text{-go}]{} \textbf{error}; \widehat{\sigma}'}$$

As observed, the translation of the traversal rules with memoization was done fairly systematically. This provides hope that the presented trace memoization technique can be automated for operational semantics rules of certain shapes, extending the technique presented by Bodin et al. [2015].

Soundness of Abstract Interpreter

The main goal of abstract interpretation is to provide *sound* analyses, where the properties shown on the approximated semantics can be depended upon to hold for the concrete semantics. For our particular abstract interpreter, the main meta-property we want to be satisfied is that the abstract expression evaluation produces values that truly over-approximate the results from all possible concrete evaluations. Formally, this can be stated as:

Theorem 7.8 (Soundness)**.** *For all valid expressions e, concrete stores σ and over-approximating abstract stores $\widehat{\sigma}$, so $\sigma \in \gamma_{\widehat{\text{Store}}}(\widehat{\sigma})$, where we have a concrete evaluation derivation $e;\widehat{\sigma} \xRightarrow[\text{expr}]{} vres;\sigma'$ and corresponding abstract evaluation derivation $e;\widehat{\sigma} \xRightarrow[\textit{a-expr}]{} \widehat{Res}$ then it holds that the abstract result set properly over-approximates the concrete result value result and store, i.e., $(vres,\sigma') \in \gamma_{RS}(\widehat{Res})$.*

7.5 Evaluation

The objective of our evaluation is to examine whether our presented abstract interpretation technique works on a wide variety of the transformations written in Rascal.

Research Questions To fulfill our objective, we aim to answer the following research questions:

- Is it possible to abstractly interpret realistic Rascal programs using type and inductive shape analysis, to verify properties of interest?
- What interesting properties require more than inductive shapes to capture, and what extensions are needed?

Subject Selection Our selection of subject transformations is based on three criteria:

1. the evaluated transformations must be realistic and preferrably from independent code bases,
2. they must be translateable to our supported subset of Rascal,
3. they must exercise important Rascal constructs like visitors and the expressive pattern matching capabilities.

The first criteria is there to reduce experimental bias from the authors, the second criteria is there to ensure that our tool is able to succesfully run on the subjects, and the third is there to ensure that we are not testing trivial or non-interesting transformations.

Subject Transformations We consider four subject transformations in our evaluation, where two of the transformations are extracted directly from real open source Rascal projects. The source code of the transformations is included in Appendix G, but we will discuss them in a high-level fashion here.

NNF The Negation Normal Form transformation [Harrison, 2009, Section 2.5] is a classical transformation in automated reasoning, which translates propositional formulae to a form that has conjunction and disjunction as the only connectives (e.g., no implication), and where negations are only applied to atomic propositions.

RSF The Rename Struct Field refactoring changes the name of a field in a struct, and correspondingly ensures that all field access expressions of that field are renamed correctly as well.

DSO0 The Desugar Oberon-0 transformation, translates for-loops and switch-statements—which are seen as syntactic sugar—to while-loops and nested if-statements respectively. The transformation is part of the Oberon-0 [Wirth, 1996] implementation in Rascal [Basten et al., 2015], which contains all the necessary stages for compiling a structured imperative programming language.

G2PE Glagol[7] is a DSL for REST-like web development, generating PHP code as output. The Glagol to PHP Expression transpilation generates PHP code for input Glagol expressions.

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

The size and target verification properties for each transformation are presented in Table Table 7.1. While the size of the transformation varies, it should be noted that they all use the expressive high-level features in Rascal and so are significantly more succinct than comparable code in non-high-level transformation languages.

The properties checked mainly focus that the output of each transformation satisfies required shape constraints. For the NNF transformation we check that the output indeed is in negation normal form, so it does not have implications (P1) and that all negations are in front of atoms (P2). For the structure field renaming transformation, we focus on properties that show that the old field has correctly been renamed to the new field name and so the old field name is not defined in a structure anymore (P3) and similarly with all field accesses (P4); note, that we need to convert field names to finite enumerations—with constructors representing the old field name, new field name and other names—instead of strings in order for our executor to be able to infer required properties. For the Oberon-0 desugaring, we check that there are no syntactic sugar constructs in the output, i.e. for-loops (P5) and switch-statements (P6), and that auxiliary data values used intermediately in the transformation are eliminated in the output (P7). Finally, we check that the Glagol to PHP translation works as expected, by checking that given inputs with a particular shape the output has a similar shape, e.g. that translating simple arithmetic-logic expressions produces similar arithmetic expressions in PHP (P8)—with no object creation, field or array access, or method invokation—and that Glagol expressions not using negation or unary plus produce no unary PHP expressions.

Threats to Validity We have selected a sample of available Rascal transformations to evaluate on, and so it can be hard to generalize whether our technique is successful for other transformations. We mitigated this by selecting our subject transformations such that they are from realistic projects and vary in authors, programming style and purpose.

We have ported our subject transformations to the Rascal Light subset in a self-contained manner to be readily executed by our tool; this process could potentially introduce some minor differences in the semantics. We mitigated this by only performing minimal changes required to the code, and ensuring that the changes done do not critically affect the credibility of our overall abstract interpretation technique.

Implementation

We have implemented our abstract interpretation technique[8] as a prototype tool for the complete Rascal Light subset presented in Chapter 6. The abstract semantics of the additional constructs follows is implemented using the process described in Section 7.4; the major challenges included fine-tuning the trace memoization strategies to the different looping constructs and functions (to handle recursion), handling extended result state with more control flow constructs, and taking into account possibly undefined values (■) during evaluation. Using types for partitioning the input during trace memoization can be inprecise, so we use a more precise partitioning strategy when needed, which for data refinements additionally uses the set of present constructors.

[8]Currently with a simple one-value domain for integers instead of intervals, which were not strictly needed for our evaluation subjects.

Transformation	LOCs	Target Property	
NNF	15	P1	Implication is not used as a connective in the result
		P2	All negations in the result are in front of atoms
RSF	35	P3	Structures should not define a fields with the old field name
		P4	There should not be any field access expression to the old field name
DSO0	125	P5	For-loops should be correctly desugared to while-loops
		P6	Switch-statements should be correctly desugared to if-statements
		P7	No auxiliary data in output
G2PE	350	P8	Only have simple PHP expressions in output given simple Glagol expressions in input
		P9	Not have unary PHP expressions if there were no sign markers or negations in the input Glagol expression

Table 7.1: Subject Transformations Statistics and Target Properties

Results

We ran our abstract interpretation tool on the evaluation subjects using Scala 2.12.2 on a 2012 Core i5 MacBook Pro. We present an overview of the results in Table 7.2 detailing the runtime and verified properties; the complete output of each run is included in Appendix H.

The runtime of the analysis is reasonable given the expressiveness of the available features, running around 4-39 seconds; separate runtimes for the G2PE transformation were included because the shape of input varies between the specified properties. There is room for improving performance, since during fixed-point calculation subterms are re-evaluated despite providing the same results.

Transformation	Runtime (s)	Property	Verified
NNF	31	P1	✓
		P2	✓
RSF	6	P3	✗
		P4	✓
DSO0	36	P5	✓
		P6	✓
		P7	✗
G2PE	4	P8	✓
	39	P9	✓

Table 7.2: Results of abstractly interpreting subject programs. Runtime is median of three runs.

All the specified target properties were succesfully verified, except for two where the result shape was not precise enough; we provide a further discussion later to describe possible extensions that would allow enough precision to capture these properties.

As an example of the the results, we show the inferred output shape for our NNF transformation in Figure 7.5. To check that our properties were satisfied, we simply had to check that implication is not part of the set of available constructors (P1), and that the negation constructor only allows atoms as subformulae instead of any general formula (P2).

Discussion

There were two properties, P3 and P7, which could not be verified from the resulting shape of the inferred inductive data refinements. Using

```
refine Formula#nnf = and(Formula#nnf, Formula#nnf) | atom(str)
                   | neg(atom(str)) | or(Formula#nnf, Formula#nnf)
```

Figure 7.5: Inferred shape for Negation Normal Form transformation

these properties as a starting point, we discuss three possible extensions to our current abstract domain that could help us get more precise results: relational constraints, abstract attributes and complement domains.

Relational Constraints Relational abstract interpreteration [Mycroft and Jones, 1985] allows specifying constraints that relate values across different variables, and even inside and across substructures [Chang and Rival, 2008; Halbwachs and Péron, 2008; Liu and Rival, 2017]. This allows both maintaining a greater precision across results, and inferring properties that are non-local, e.g., that two values not only have similar shape but are actually equal.

Figure 7.6 shows the inferred refinement for the structure field renaming transformation: we can see that the structures contained in target packages remain unrefined and thus P3 does not hold since old field name should not be in definitions. We assume that structures and fields are stored in the containing maps using their names as key, but without relational constraints it is not possible to express this property; this disallows strong map updates and the update to the field definitions is thus not captured. Relational properties would also allow supporting the version of the refactoring that works on classes and additionally takes typing into account.

Abstract Attributes Property P7 provides more interesting verification challenges that pertain particularly to transformations. Recall that the property concerns proving elimination of auxiliary data from the result; concretely the auxiliary statement begin is used to intermediately allow returning a list of statements where a single statement is required, and a separate pass is used to flatten the contained list of statements contained with the surrounding list of statements (see Figure 7.7).

Verifying properties that only hold after fixed-point iteration, seems suitably tackled using *abstract attributes* which extract additional information about target structures. For this particular property, a generalization of the multiset abstraction suggested by Perrelle and Halbwachs

```
refine Package#rnf =
  package(map[str, Struct], map[str, Function#rnf])
data Struct = struct(str name, map[Nominal, Field] fields);
data Field = field(Nominal name, str typ);
refine Function#rnf =
  function(str, str, list[Parameter], Stmt#rnf)
refine Stmt#rnf =
  assignstmt(Expr#rnf, Expr#rnf) | block(list[Stmt#rnf])
| ifstmt(Expr#rnf, Stmt#rnf, Stmt#rnf) | returnstmt(Expr#rnf)
refine Expr#rnf =
  fieldaccessexpr(Expr#rnf, Nominal#rnf)
| functioncallexpr(Expr#rnf, str, list[Expr#rnf]) | varexpr(str)
data Nominal = nfn() | ofn() | other();
refine Nominal#rnf = nfn() | other()
```

Figure 7.6: Inferred shape for Rename Struct Field transformation

```
public list[Statement] flattenBegin(list[Statement] stats) {
    return innermost visit (stats) {
        case [*Statement s1, begin(b), *Statement s2] =>
                s1 + b + s2
    }
}
```

Figure 7.7: Flattening auxiliary begin statement

[2010] for data types, could be useful to track the number of elements of various sub-elements, e.g., the number of begin statements. Using techniques from term rewriting [Dershowitz and Manna, 1979] one can then show a decrease in number of elements each iteration, inferring that they get eliminated when the fixed-point is reached. Other abstract attributes—such as suggested by Bouajjani et al. [2012] and Pham and Whalen [2013]—can capture other properties of interest like contained set of elements or size of a data type, and in conjunction with relational constraints show meaningful relations between input and output structures, e.g., that the output uses the same variables as the input.

Complement Domains The output for DSO0 and G2PE additionally produced false positive errors, due to values being inferred as potentially undefined ■. The main culprit was that our analysis did not precisely enough capture complementary information—i.e. absence of type or shape—for non-data type values during pattern matching, and so some **switch**-statements fell-through. This can be solved by having a complementary domain component of disallowed types and values for pattern matching, similarly to what we had for TRON in Chapter 5.

7.6 Related Work

Data and Shape Domains Our discussion presented future directions that combine our inductive refinement with ideas from existing work that support more expressive constraints. Bouajjani et al. [2012] allows specifying a flexible set of abstract attributes on lists, both strucural such as acylicity and data-relational such as equality of contained multi-sets of elements. Techniques suggested by Chang and Rival [2008], Vazou et al. [2013], and Albarghouthi et al. [2015] 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.

Modular Program Analysis Cousot and Cousot [2002] 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 [Padon et al., 2016]. Calcagno et al. [2011] suggests using *bi-abduction* for compositional analysis of programs, but their framework is primarily focused towards pointer programs with only a fixed set of non-nested data types supported. Toubhans et al. [2013] and Rival et al. [2014] further develops the ideas of modular domain design for pointer-manipulating programs supporting a rich set of data abstractions; our domain construction focuses more on pure data-structures with a more heterogeneous construction—i.e., where we have differents kinds of algebraic data types, collections, and basic values—still allowing modular and automated inference of inductive refinements for these types.

Verification of Transformations There are various approaches [Jackson et al., 2011; Büttner et al., 2012; Wang et al., 2014] in verification of model transformations that encode transformations as declarative formulae for automated solvers, but such techniques only work for transformations of limited expressiveness and are due to the black-box nature of automated solvers hard to extend. Techniques for model transformation verification based on static analysis such as in Cuadrado et al. [2017] scale to more expressive transformation features, but are currently focused on verification of rule errors based on types and undefinedness.

Semantic typing has been used by Castagna and Nguyen [2008] and Benzaken et al. [2013] to infer recursive type and shape properties for language with high-level constructs for querying and iteration. The languages considered are small calculi with limited expressiveness compared to the fully feature transformation language subset of Rascal we consider, we have a more flexible domain and abstract interpreter design, and an extensive evaluation showing that our works well with realistic Rascal programs.

Another way to verify transformations using abstract interpretation has been presented by Rival [2004], which uses translation validation and 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 very rich semantic properties but only for concrete input, while our technique verifies simpler properties as shapes but for all possible input.

7.7 Recap

We have presented a formal Schmidt-style abstract interpreter for Rascal Light, that supports verifying type and inductive shape properties. Our interpreter supported all the required high-level transformation features including traversals, backtracking, exceptions and fixed-point iteration. We further presented a practical technique for modularly constructing abstract domains and adapted the idea of *trace memoization* to work with data from infinite abstract domains by using finite input-determined partitioning to widen input on recursion and ensure termination of our algorithm. Finally, we evaluated our algorithm on a variety of realistic transformations showing that it is possible to efficiently and effectively verify inductive shape properties of interest.

Chapter 8

Conclusion

The primary goal for this dissertation was to use traditional techniques in programming languages as a basis for developing foundational theories and tools for transformation languages with high-level features such as traversals, backtracking and fixed-point iteration. The goal of this thesis has been successfully reached by a series of contributions (C1-C5) which together fulfilled the stated objectives (O1-O4).

In Chapter 3, I presented a detailed analysis of a wide selection of declarative transformation languages (C1) using available literature to identify key features and clarify how they contributed to the target languages' computational expressiveness (O1). My analysis showed that all examined declarative transformation languages, except the bidirectional ones, were Turing-complete and so could not be treated specially with regards to verification. This compelled me to focus on general purpose high-level transformation languages like TXL and Rascal, which had the advantage of having and explicit control flow and more commonalities with existing programming languages, making it easier to adapt target programming language verification techniques.

Chapter 4 presented a joint effort to design and validate an industrial transformation (C2) modernizing an industrial configuration tool from a C++ imperative implementation to logical constraints. The effort contributed to understanding how the high-level transformation features were used in practice, and provided first-hand experience with translation validation and symbolic execution (O2). A major concrete outcome was that it displayed the necessity of formal verification techniques for transformations, catching 50 serious bug cases in an expert-written otherwise well-tested transformation.

To allow for effective automated white-box testing of transformations (O3), I had in Chapter 5 developed a formal symbolic execution algorithm for a core transformation language, TRON, which captured key high-level transformation language features such including deep type-directed traversal (C3). The algorithm was implemented as a prototype tool and evaluated using a set of realistic refactorings and model transformations against a baseline black-box algorithm optimizing for litterature-suggested metrics; the evaluation showed that our algorithm compared very favorably to the baseline black-box algorithm, achieving almost full coverage on all subject programs, albeit at a cost of slower—but still reasonable—execution time.

I formalized an extensive subset of the operational part of the Rascal transformation language—called Rascal Light—in Chapter 6, which is the first formalization for a large subset of the Rascal language we know of (C4). The formalization has been based on the Rascal implementation, documentation and personal correspondence with developers, it has been implemented as a prototype and tested with a set of Rascal programs, and a series of semantic properties regarding strong typing, safety and termination have been formally proven. The availability of this formalization allows principled development of foundational verification techniques including abstract interpretation-based static analysis (O4), and other complementary semantics like a type system or axiomatic semantics.

Finally, I implemented a Schmidt-style abstract interpreter (C5) for Rascal in Chapter 7 that allowed verification of types and inductive shape properties (O4). The work provided a practical path towards modular design of abstract domains, and an adaption of the idea of *trace memoization*—which allows discovering fixed-points in operational semantics—to support working precisely with input from infinite domains, by utilising input-determined finite partitioning. The technique has been implemented as a prototype tool, which have been evaluated on realistic transformations of different kinds written in Rascal, and was able to infer complex shape properties such as that the output of the negation normal form transformation only had negations in front of atoms, and that a transpilation of an expression with no negative or positive sign operators produced no output unary PHP expressions.

Bibliography

Aditya Agrawal, Zsolt Kalmar, Gabor Karsai, Feng Shi, and Attila Vizhanyo. *GReAT User Manual*, 2003. URL http://www.escherinstitute.org/Plone/tools/suites/mic/great/GReAT%20User%20Manual.pdf.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 2nd edition, 1986a. ISBN 0-201-10088-6. URL http://www.worldcat.org/oclc/12285707.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 8. In *Addison-Wesley series in computer science / World student series edition* Aho et al. [1986a], 2nd edition, 1986b. ISBN 0-201-10088-6. URL http://www.worldcat.org/oclc/12285707.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 9-11. In *Addison-Wesley series in computer science / World student series edition* Aho et al. [1986a], 2nd edition, 1986c. ISBN 0-201-10088-6. URL http://www.worldcat.org/oclc/12285707.

Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, pages 427–447, 1991. URL https://doi.org/10.1007/3540543961_21.

Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information & Software Technology*, 49(3):275–291, 2007. doi: 10.1016/j.infsof.2006.10.012. URL http://dx.doi.org/10.1016/j.infsof.2006.10.012.

Ahmad Salim Al-Sibahi. On the computational expressiveness of model transformation languages. *IT University. Technical Report Series*, January 2015. ISSN 1600-6100.

Ahmad Salim Al-Sibahi. The Formal Semantics of Rascal Light. *CoRR*, abs/1703.02312, 2017a. URL `http://arxiv.org/abs/1703.02312`.

Ahmad Salim Al-Sibahi. Abstract interpretation of high-level transformations. In *Student Research Competition at 44th ACM SIGPLAN Symposium on Principles of Programming Languages 2017, POPL SRC 2017, Paris, France, January 15-21, 2017*, 2017b. Poster.

Ahmad Salim Al-Sibahi, Aleksandar Dimovski, and Andrzej Wąsowski. SymexTRON: Symbolic execution of high-level transformation languages: Symbolic execution of high-level transformations. *IT University. Technical Report Series*, September 2016. ISSN 1600-6100.

Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wąsowski. Symbolic execution of high-level transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands*, pages 207–220, 2016. doi: 10.1145/2997364.2997382.

Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Thomas P. Jensen, and Andrzej Wąsowski. Verifying Transformations using Inductive Shape Analysis. 2017.

Jesse Alama. The lambda calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2014 edition, 2014.

Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. Spatial interpolants. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 634–660, 2015. doi: 10.1007/978-3-662-46669-8_26. URL `https://doi.org/10.1007/978-3-662-46669-8_26`.

Freddy Allilaire and Frédéric Jouault. Families to persons: A simple illustration of model-to-model transformation, 2007. URL `https://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf`.

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679, 2017. URL `http://dl.acm.org/citation.cfm?id=3009866`.

Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 921–928, 2012. doi: 10.1109/ICST.2012.197. URL `https://doi.org/10.1109/ICST.2012.197`.

Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. ISBN 978-0-521-45520-6.

Roland Carl Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors. *Datatype-Generic Programming - International Spring School, SSDGP 2006, Nottingham, UK, April 24-27, 2006, Revised Lectures*, volume 4719 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-76785-5. doi: 10.1007/978-3-540-76786-2. URL `https://doi.org/10.1007/978-3-540-76786-2`.

Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2007.

András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1280–1287, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. doi: 10.1145/1141277.1141575. URL `http://doi.acm.org/10.1145/1141277.1141575`.

Hans Barengdt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 2 edition, 1985. ISBN 978-0444867483.

Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. Dsltrans: A turing incomplete transformation language. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - Third International Conference, SLE 2010,* *Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 2010. ISBN 978-3-642-19439-9. doi: 10.1007/978-3-642-19440-5_19. URL `http://dx.doi.org/10.1007/978-3-642-19440-5_19`.

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. Modular language implementation in rascal - experience report. *Sci. Comput. Program.*, 114:7–19, 2015. URL `http://dx.doi.org/10.1016/j.scico.2015.11.003`.

I.D. Baxter, C. Pidgeon, and M. Mehlich. DMS ®: program transformations for practical scalable software evolution. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 625–634, May 2004. doi: 10.1109/ICSE.2004.1317484.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. 10(4):265–289, 2003. ISSN 1236-6064.

Jon Louis Bentley. Little languages. *Commun. ACM*, 29(8):711–721, 1986. doi: 10.1145/6424.315691. URL `http://doi.acm.org/10.1145/6424.315691`.

Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of nosql languages. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 101–114, 2013. doi: 10.1145/2429069.2429083. URL `http://doi.acm.org/10.1145/2429069.2429083`.

Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. In *SAT Competition 2014, Vienna, Austria, July 14-17, Proceedings*, page 2, 2014.

G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.

D Bisztray and Reiko Heckel. Combining Termination Criteria by Isolating Deletion. *Graph Transformations*, pages 203–217, 2010. URL `http://link.springer.com/chapter/10.1007/978-3-642-15928-2_14`.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009. doi: 10.1007/s10817-009-9148-3. URL `http://dx.doi.org/10.1007/s10817-009-9148-3`.

Jakob Blumer, Rubino Geiß, and Edgar Jakumeit. *The GrGen.NET User Manual*, oct 2010. URL `https://pp.info.uni-karlsruhe.de/svn/grgen-public/trunk/grgen/doc/grgen.pdf`.

Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 29–40, 2015. doi: 10.1145/2676724.2693174. URL `http://doi.acm.org/10.1145/2676724.2693174`.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 407–419, 2008. doi: 10.1145/1328438.1328487. URL `http://doi.acm.org/10.1145/1328438.1328487`.

Egon Börger and Robert F Stärk. *Abstract State Machines: A Method for High-level System Design and Analysis*. Springer, 2003.

Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15:55–70, 1998. doi: 10.1016/S1571-0661(05)82552-6. URL `https://doi.org/10.1016/S1571-0661(05)82552-6`.

Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 1–22, 2012. doi: 10.1007/978-3-642-27940-9_1. URL `https://doi.org/10.1007/978-3-642-27940-9_1`.

Peter Braun and Frank Marschall. BOTL - the bidirectional object oriented transformation language. Technical report, Technische Universität München, 2003.

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

Stefan Bunzel. AUTOSAR - the standardized software architecture. *Informatik Spektrum*, 34(1):79–83, 2011. doi: 10.1007/s00287-010-0506-7. URL `http://dx.doi.org/10.1007/s00287-010-0506-7`.

Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 432–448. Springer, 2012. ISBN 978-3-642-33665-2. doi: 10.1007/978-3-642-33666-9_28. URL `http://dx.doi.org/10.1007/978-3-642-33666-9_28`.

Cristian Cadar and Alastair F. Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 765–768, 2016. doi: 10.1145/2889160.2889206. URL `http://doi.acm.org/10.1145/2889160.2889206`.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. ISBN 978-1-931971-65-2. URL `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi: 10.1145/2049697.2049700. URL `http://doi.acm.org/10.1145/2049697.2049700`.

Daniel Calegari and Nora Szasz. Verification of model transformations: A survey of the state-of-the-art. *Electr. Notes Theor. Comput. Sci.*, 292: 5–25, 2013. doi: 10.1016/j.entcs.2013.02.002. URL `https://doi.org/10.1016/j.entcs.2013.02.002`.

Giuseppe Castagna and Kim Nguyen. Typed iterators for XML. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 15–26, 2008. doi: 10.1145/1411204.1411210. URL `http://doi.acm.org/10.1145/1411204.1411210`.

María Victoria Cengarle and Alexander Knapp. Ocl 1.4/5 vs. 2.0 expressions formal semantics and expressiveness. *Software and Systems Modeling*, 3(1):9–30, 2004. ISSN 1619-1366. doi: 10.1007/s10270-003-0035-9. URL `http://dx.doi.org/10.1007/s10270-003-0035-9`.

Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 247–260, 2008. doi: 10.1145/1328438.1328469. URL `http://doi.acm.org/10.1145/1328438.1328469`.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14, 2010. doi: 10.1145/1863543.1863547. URL `http://doi.acm.org/10.1145/1863543.1863547`.

Magnus Christerson, David Frankel, and Todd Schiller. Financial domain-specific language listing, October 2013. URL `http://www.webcitation.org/query?url=http%3A%2F%2Fwww.dslfin.org%2Fresources.html&date=2017-07-24`.

Horatiu Cirstea and Claude Kirchner. The rewriting calculus - part I. *Logic Journal of the IGPL*, 9(3):339–375, 2001a. doi: 10.1093/jigpal/9.3.339. URL `https://doi.org/10.1093/jigpal/9.3.339`.

Horatiu Cirstea and Claude Kirchner. The rewriting calculus - part II. *Logic Journal of the IGPL*, 9(3):377–410, 2001b. doi: 10.1093/jigpal/9.3.377. URL `https://doi.org/10.1093/jigpal/9.3.377`.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279, 2000. doi:

10.1145/351240.351266. URL http://doi.acm.org/10.1145/351240.351266.

Robert Clarisó, Jordi Cabot, Esther Guerra, and Juan de Lara. Backwards reasoning for model transformations: Method and applications. *Journal of Systems and Software*, 116:113–132, 2016. doi: 10.1016/j.jss.2015.08.017. URL https://doi.org/10.1016/j.jss.2015.08.017.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-71940-3. doi: 10.1007/978-3-540-71999-1. URL https://doi.org/10.1007/978-3-540-71999-1.

B. Jack Copeland. The church-turing thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2015 edition, 2015. URL https://plato.stanford.edu/archives/sum2015/entries/church-turing/.

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

Ricardo Corin and Felipe Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, volume 6542 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2011. ISBN 978-3-642-19124-4. doi: 10.1007/978-3-642-19125-1_5. URL http://dx.doi.org/10.1007/978-3-642-19125-1_5.

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

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium* *on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977. doi: 10.1145/512950.512973. URL `http://doi.acm.org/10.1145/512950.512973`.

Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 170–181, 1995. URL `http://doi.acm.org/10.1145/224164.224199`.

Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 159–178, 2002. doi: 10.1007/3-540-45937-5_13. URL `https://doi.org/10.1007/3-540-45937-5_13`.

Arlen Cox, Bor-Yuh Evan Chang, Huisong Li, and Xavier Rival. Abstract domains and solvers for sets reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 356–371, 2015. doi: 10.1007/978-3-662-48899-7_25. URL `https://doi.org/10.1007/978-3-662-48899-7_25`.

J. Sanchez Cuadrado, E. Guerra, and J. de Lara. Static analysis of model transformations. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2635137.

CWI Amsterdam. Abstract syntax tree of muRascal. online, February 2017. URL `https://web.archive.org/web/20170726155735/https://raw.githubusercontent.com/usethesource/rascal/master/src/org/rascalmpl/library/experiments/Compiler/muRascal/AST.rsc`.

Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006. doi: 10.1147/sj.453.0621. URL `https://doi.org/10.1147/sj.453.0621`.

Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom3. *Software and Systems Modeling*, 3(3):194–209, 2004. ISSN 1619-1366. doi: 10.1007/s10270-003-0047-5. URL `http://dx.doi.org/10.1007/s10270-003-0047-5`.

Xianghua Deng, Jooyong Lee, and Robby. Efficient and formal generalized symbolic execution. *Autom. Softw. Eng.*, 19(3):233–301, 2012. doi: 10.1007/s10515-011-0089-9. URL `http://dx.doi.org/10.1007/s10515-011-0089-9`.

Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979. doi: 10.1145/359138.359142. URL `http://doi.acm.org/10.1145/359138.359142`.

Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-based model checking without a family-based model checker. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, pages 282–299, 2015a. doi: 10.1007/978-3-319-23404-5_18.

Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-based model checking using off-the-shelf model checkers: extended abstract. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, page 397, 2015b. doi: 10.1145/2791060.2791119. URL `http://doi.acm.org/10.1145/2791060.2791119`.

Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Efficient family-based model checking via variability abstractions. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2016. ISSN 1433-2787. doi: 10.1007/s10009-016-0425-2. URL `http://dx.doi.org/10.1007/s10009-016-0425-2`.

Keith Duddy, Michael Lawley, and Sridhar Iyengar. Mof query/views/-transformations: Second revised submission. Technical report, 2004. URL `http://tefkat.sourceforge.net/publications/ad-04-01-06.pdf`.

Levent Erkök and John Matthews. High assurance programming in cryptol. In *Fifth Cyber Security and Information Intelligence Research Workshop, CSIIRW '09, Knoxville, TN, USA, April 13-15, 2009*, page 60, 2009. doi: 10.1145/1558607.1558676. URL `http://doi.acm.org/10.1145/1558607.1558676`.

Antonio Estévez, Javier Padrón, E. Victor Sánchez Rebull, and José L. Roda. ATC: A low-level model transformation language. In *Model-Driven Enterprise Information Systems, Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006, In conjunction with ICEIS 2006, Paphos, Cyprus, May 2006*, pages 64–74, 2006.

Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2014. ISBN 012415817X, 9780124158177.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN 978-0-262-06275-6. URL `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885`.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 113–128, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.113. URL `https://doi.org/10.4230/LIPIcs.SNAPL.2015.113`.

Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Thomas Degueule. Using meta-model coverage to qualify test oracles. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013. URL `http://ceur-ws.org/Vol-1077/amt13_submission_3.pdf`.

Thorsten Fischer, J"org Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67203-6. doi: 10.1007/978-3-540-46464-8_21. URL `http://dx.doi.org/10.1007/978-3-540-46464-8_21`.

Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN 978-

0-201-48567-7. URL `http://martinfowler.com/books/refactoring.html`.

Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011. ISBN 978-0-321-71294-3. URL `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html`.

Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277, 1991. URL `http://doi.acm.org/10.1145/113445.113468`.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005. doi: 10.1145/1065010.1065036. URL `http://doi.acm.org/10.1145/1065010.1065036`.

C Gomes, Bruno Barroca, and Vasco Amaral. Classification of Model Transformation Tools: Pattern Matching Techniques. *Model-Driven Engineering Languages and Systems*, pages 619–635, 2014. URL `http://link.springer.com/chapter/10.1007/978-3-319-11653-2_38`.

Carlos A. González and Jordi Cabot. Atltest: A white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pages 449–464, 2012. doi: 10.1007/978-3-642-33666-9_29. URL `http://dx.doi.org/10.1007/978-3-642-33666-9_29`.

James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, 8th edition, 2014. ISBN 978-0133900699.

Jim Gray. Why do computers stop and what can be done about it? In *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*, pages 3–12. IEEE Computer Society, 1986. ISBN 0-8186-0690-8.

Yuri Gurevich. Evolving algebras 1993: Lipari guide. *Specification and validation methods*, pages 9–36, 1995.

Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 339–348, 2008. doi: 10.1145/1375581.1375623. URL `http://doi.acm.org/10.1145/1375581.1375623`.

John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. ISBN 978-0-521-89957-4.

Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C$^\sharp$ Programming Language*. Addison-Wesley, 4th edition, 2011. ISBN 978-0321741769.

Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling*, 15(3):907–928, 2016. doi: 10.1007/s10270-014-0450-0. URL `https://doi.org/10.1007/s10270-014-0450-0`.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL `http://doi.acm.org/10.1145/363235.363259`.

C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, pages 78–78, 2005. doi: 10.1007/978-3-540-30579-8_5. URL `http://dx.doi.org/10.1007/978-3-540-30579-8_5`.

J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007. ISBN 9780321455369. URL `http://books.google.dk/books?id=6ytHAQAAIAAJ`.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008. doi: 10.1007/s10990-008-9025-5. URL `https://doi.org/10.1007/s10990-008-9025-5`.

Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996. doi: 10.1145/242224.242477. URL `http://doi.acm.org/10.1145/242224.242477`.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 435–445, 2014. doi: 10.1145/2568225.2568271. URL `http://doi.acm.org/10.1145/2568225.2568271`.

Alexandru Florin Iosif-Lazăr, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wąsowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA*, pages 597–607, 2015. doi: 10.1109/ASE.2015.84.

Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pages 653–667, 2011. doi: 10.1007/978-3-642-24485-8_48. URL `https://doi.org/10.1007/978-3-642-24485-8_48`.

Edgar Jakumeit. Erweiterung der regelsprache eines graphersetzungswerkzeugs um rekursive regeln mittels sterngraphgrammatiken und paragraphgrammatiken. Diploma thesis, Universität Karlsruhe, jul 2008.

Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net. *International Journal on Software Tools for Technology Transfer*, 12(3/4):263 – 271, 2010. ISSN 14332779. URL `http://search.ebscohost.com/login.aspx?direct=true&db=aph&AN=51523140&site=ehost-live`.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN 978-0-13-020249-9.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

doi: 10.1016/j.scico.2007.08.002. URL `http://dx.doi.org/10.1016/j.scico.2007.08.002`.

Gilles Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, pages 22–39, 1987. doi: 10.1007/BFb0039592. URL `https://doi.org/10.1007/BFb0039592`.

Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 444–463. ACM, 2010. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. URL `http://doi.acm.org/10.1145/1869459.1869497`.

Michael Kay. XSL transformations (XSLT) version 3.0. W3C recommendation, W3C, June 2017. https://www.w3.org/TR/2017/REC-xslt-30-20170608/.

Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 553–568, 2003. doi: 10.1007/3-540-36577-X_40. URL `http://dx.doi.org/10.1007/3-540-36577-X_40`.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`.

Stephen Cole Kleene. *Mathematical Logic*. John Wiley & Sons, 1 edition, 1967. ISBN 978-0471490333.

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177, 2009. doi: 10.1109/SCAM.2009.28. URL `https://doi.org/10.1109/SCAM.2009.28`.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Metaprogramming with Rascal. In JoãoM. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-18022-4. doi: 10.1007/978-3-642-18023-1_6. URL `http://dx.doi.org/10.1007/978-3-642-18023-1_6`.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, pages 46–60, 2008. doi: 10.1007/978-3-540-69927-9_4. URL `https://doi.org/10.1007/978-3-540-69927-9_4`.

Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, and Richard Paige. The epsilon book. September 2014. URL `http://eclipse.org/epsilon/doc/book/`.

B. Korel. A dynamic approach of test data generation. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 311–317, Nov 1990. doi: 10.1109/ICSM.1990.131379.

Alexander Kraas. Realizing model simplifications with QVT operational mappings. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, pages 53–62, 2014. URL `http://ceur-ws.org/Vol-1285/paper06.pdf`.

Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE. Vol. 22. 2009*, 2009.

Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665. URL `http://dx.doi.org/10.1109/CGO.2004.1281665`.

Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Bidirectional model transformation with precedence triple graph grammars. In *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, pages 287–302, 2012. doi: 10.1007/978-3-642-31491-9_22. URL `https://doi.org/10.1007/978-3-642-31491-9_22`.

Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-31780-7. doi: 10.1007/11663430_15. URL `http://dx.doi.org/10.1007/11663430_15`.

László Lengyel, Tihamér Levendovszky, and Hassan Charaf. A visual control flow language and its termination properties. *International Journal of Computer, Information, Systems and Control Engineering*, 1(8):2505 – 2510, 2007. ISSN 1307-6892. URL `http://waset.org/Publications?p=8`.

Xavier Leroy. The Caml Light system, documentation and user's guide 0.74. `https://caml.inria.fr/pub/docs/manual-caml-light/`, December 1997.

Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65 – 75, 2005. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2004.12.040. URL `http://www.sciencedirect.com/science/article/pii/S1571066105001155`. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004) Graph-Based Tools 2004.

Andriy Levytskyy and Eugene J. H. Kerckhoffs. From class diagrams to zope products with the meta-modelling tool atom. *Summer Computer Simulation Conference*, pages 295–300, 2003.

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

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015. doi: 10.1145/2644805. URL `http://doi.acm.org/10.1145/2644805`.

Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 1–12, 2011. doi: 10.1145/2036918.2036920. URL `http://doi.acm.org/10.1145/2036918.2036920`.

Levi Lucio and Hans Vangheluwe. Symbolic Execution for the Verification of Model Transformations. *VOLT@STAF*, pages 1–29, April 2013.

Simon Marlow. *The Haskell 2010 Language Report*. 2010. URL `https://www.haskell.org/definition/haskell2010.pdf`.

Tom Mens and P Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, 2006. URL `http://www.sciencedirect.com/science/article/pii/S1571066106001435`.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. doi: 10.1145/1118890.1118892. URL `http://doi.acm.org/10.1145/1118890.1118892`.

Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 49–60, 2007. doi: 10.1145/1291201.1291208. URL `http://doi.acm.org/10.1145/1291201.1291208`.

Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling testing of refactoring engines. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 371–380, 2014. doi: 10.1109/ICSME.2014.59. URL `http://dx.doi.org/10.1109/ICSME.2014.59`.

Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985*, pages 156–171, 1985. doi: 10.1007/3-540-16446-4_9. URL `https://doi.org/10.1007/3-540-16446-4_9`.

Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A Survey on Domain-Specific Languages in Robotics. *Journal of Software Engineering for Robotics*, 8810(Chapter 17):195–206, October 2014.

Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*. June 2016. version 1.3.

Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014. doi: 10.1145/2591013. URL `http://doi.acm.org/10.1145/2591013`.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In *Proceedings of the 11th International Conference on Implementation and Application of Automata*, CIAA'06, pages 275–276, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-37213-X, 978-3-540-37213-4. doi: 10.1007/11812128_26. URL `http://dx.doi.org/10.1007/11812128_26`.

Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 217–231, 2016. doi: 10.1145/2837614.2837640. URL `http://doi.acm.org/10.1145/2837614.2837640`.

Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013. doi: 10.1007/s10515-013-0122-2. URL `http://dx.doi.org/10.1007/s10515-013-0122-2`.

S Patzina and Lars Patzina. A case study based comparison of atl and sdm. *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, 7233, 2012. doi: 10.1007/978-3-642-34176-2. URL http://link.springer.com/chapter/10.1007/978-3-642-34176-2_18http://www.springerlink.com/index/10.1007/978-3-642-34176-2.

Valentin Perrelle and Nicolas Halbwachs. An analysis of permutations in arrays. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, pages 279–294, 2010. doi: 10.1007/978-3-642-11319-2_21. URL https://doi.org/10.1007/978-3-642-11319-2_21.

Tuan-Hung Pham and Michael W. Whalen. An improved unrolling-based decision procedure for algebraic data types. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, pages 129–148, 2013. doi: 10.1007/978-3-642-54108-7_7. URL https://doi.org/10.1007/978-3-642-54108-7_7.

Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

Detlef Plump. On Termination of Graph Rewriting. *Graph-Theoretic Concepts in Computer Science*, 1995. URL http://link.springer.com/chapter/10.1007/3-540-60618-1_68.

Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2), 1998.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998. ISBN 3-540-64356-7. doi: 10.1007/BFb0054170. URL http://dx.doi.org/10.1007/BFb0054170.

Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–269, April 1946. ISSN

0002-9904. doi: 10.1090/S0002-9904-1946-08555-9. URL `http://www.ams.org/journal-getitem?pii=S0002-9904-1946-08555-9`.

Emil L. Post. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic*, 12(1):1–11, 1947. URL `http://journals.cambridge.org/abstract_S0022481200076295`.

Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software and System Modeling*, 14 (2):1003–1028, 2015. doi: 10.1007/s10270-013-0358-0. URL `http://dx.doi.org/10.1007/s10270-013-0358-0`.

Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 479–485, 2003. doi: 10.1007/978-3-540-25959-6_40. URL `https://doi.org/10.1007/978-3-540-25959-6_40`.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74 (2):358–366, 1953.

Adrián Riesco. Test-case generation for maude functional modules. In *WADT 2010, Etelsen, Germany, July 1-4, 2010*, pages 287–301, 2010. doi: 10.1007/978-3-642-28412-0_18. URL `http://dx.doi.org/10.1007/978-3-642-28412-0_18`.

Adrián Riesco. Using narrowing to test maude specifications. In *WRLA'12, Tallinn, Estonia, March 24-25, 2012*, pages 201–220, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34004-8. doi: 10.1007/978-3-642-34005-5_11. URL `http://dx.doi.org/10.1007/978-3-642-34005-5_11`.

Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 1–13, 2004. doi: 10.1145/964001.964002. URL `http://doi.acm.org/10.1145/964001.964002`.

Xavier Rival, Antoine Toubhans, and Bor-Yuh Evan Chang. Construction of abstract domains for heterogeneous properties (position paper). In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 489–492, 2014. doi: 10.1007/978-3-662-45231-8_40. URL `https://doi.org/10.1007/978-3-662-45231-8_40`.

Mads Rosendahl. 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, Manhattan, Kansas, USA, 19-20th September 2013.*, pages 84–104, 2013. doi: 10.4204/EPTCS.129.7. URL `https://doi.org/10.4204/EPTCS.129.7`.

Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. ISBN 98-102288-48.

Olga Runge. *The AGG 1.5.0 Development Environment: The User Manual*, 2006. URL `http://user.cs.tu-berlin.de/~gragra/agg/AGG-ShortManual/AGG-ShortManual.html`.

John M. Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24 (9):709–720, 1998. doi: 10.1109/32.713327. URL `https://doi.org/10.1109/32.713327`.

Hanan Samet. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*, pages 492–497, 1976. doi: 10.1145/800191.805648. URL `http://doi.acm.org/10.1145/800191.805648`.

Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 67–72, 2009. doi: 10.1145/1481848.1481859. URL `http://doi.acm.org/10.1145/1481848.1481859`.

David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation*, 10(3):237–271, 1998.

Johannes Schönböck, Gerti Kappel, Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. Tetrabox - A generic white-box testing framework for model transformations. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 75–82, 2013. doi: 10.1109/APSEC.2013.21. URL `http://dx.doi.org/10.1109/APSEC.2013.21`.

Andy Schürr. PROGRES, a visual language and environment for programming with graph rewriting systems. Technical report, RWTH Aachen, 1994.

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976. URL `http://dx.doi.org/10.1137/0205037`.

Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems - Software Technologies, Engineering Processes, and Business Practices*. SEI series in software engineering. Addison-Wesley, 2003. ISBN 978-0-321-11884-4. URL `http://www.informit.com/store/modernizing-legacy-systems-software-technologies-engineering-9780321118844`.

Gehan M. K. Selim, Fabian Büttner, James R. Cordy, Jürgen Dingel, and Shige Wang. Automated verification of model transformations in the automotive industry. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 690–706. Springer, 2013. ISBN 978-3-642-41532-6. doi: 10.1007/978-3-642-41533-3_42. URL `http://dx.doi.org/10.1007/978-3-642-41533-3_42`.

Gehan M. K. Selim, Levi Lucio, James R. Cordy, Jürgen Dingel, and Bentley J. Oakes. Specification and verification of graph-based model transformation properties. In Holger Giese and Barbara König, editors, *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*, volume 8571 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2014. ISBN 978-3-319-09107-5. doi: 10.1007/978-3-319-09108-2_8. URL `http://dx.doi.org/10.1007/978-3-319-09108-2_8`.

Gehan M. K. Selim, Shige Wang, James R. Cordy, and Juergen Dingel. Model transformations for migrating legacy deployment models in the automotive industry. *Software and System Modeling*, 14(1):365–381, 2015. doi: 10.1007/s10270-013-0365-1. URL `http://dx.doi.org/10.1007/s10270-013-0365-1`.

Jim Shore. Fail fast. *IEEE Software*, 21(5):21–25, 2004. doi: 10.1109/MS.2004.1331296. URL `http://doi.ieeecomputersociety.org/10.1109/MS.2004.1331296`.

Jon Siegel, editor. *CORBA 3 fundamentals and programming*. OMG Press, John Wiley & Sons, New York, 2nd edition, 2000.

Anthony M. Sloane. Lightweight language processing in kiama. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 408–425, 2009. doi: 10.1007/978-3-642-18023-1_12. URL `https://doi.org/10.1007/978-3-642-18023-1_12`.

Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982. URL `http://dx.doi.org/10.1137/0211062`.

Eugene Syriani and Hans Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414, 2013. ISSN 1619-1366. doi: 10.1007/s10270-011-0205-0. URL `http://dx.doi.org/10.1007/s10270-011-0205-0`.

Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In JohnL. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22120-3. doi: 10.1007/978-3-540-25959-6_35. URL `http://dx.doi.org/10.1007/978-3-540-25959-6_35`.

Gabriele Taentzer. What Algebraic Graph Transformations Can Do For Model Transformations. *Electronic Communications of the EASST*, 30, 2010.

R. Gregory Taylor. *Models of Computation and Formal Languages*. Oxford University Press, Oxford, UK, 1998. ISBN 0-19-510983-X.

The Eclipse Foundation. *Xtend User Guide*, September 2014. URL `http://www.eclipse.org/xtend/documentation/2.7.0/Xtend%20User%20Guide.pdf`.

The PROGReS Developer Team. *The PROGRES Language Manual Version 9.x*, 1999. URL `http://www-i3.informatik.rwth-aachen.de/files/progres/PROGRESLanguageManual.pdf`.

Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008. doi: 10.1007/978-3-540-79124-9_10. URL `http://dx.doi.org/10.1007/978-3-540-79124-9_10`.

Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Lazy execution of model-to-model transformations. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24484-1. doi: 10.1007/978-3-642-24485-8_4. URL `http://dx.doi.org/10.1007/978-3-642-24485-8_4`.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 632–647, 2007. doi: 10.1007/978-3-540-71209-1_49. URL `http://dx.doi.org/10.1007/978-3-540-71209-1_49`.

Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 375–395, 2013. doi: 10.1007/978-3-642-35873-9_23. URL `https://doi.org/10.1007/978-3-642-35873-9_23`.

Javier Troya and Antonio Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011. doi: 10.5381/jot.2011.10.1.a5. URL `http://dx.doi.org/10.5381/jot.2011.10.1.a5`.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000. doi: 10.1145/352029.352035. URL `http://doi.acm.org/10.1145/352029.352035`.

Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. *Proceedings of the AIS '2002 conference (AI, Simulation and Planning in High Autonomy Systems)*, 2002.

Dániel Varró and Andras Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3): 214 – 234, 2007. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2007.05.004. URL `http://www.sciencedirect.com/science/article/pii/S016764230700127X`. Special Issue on Model Transformation.

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013. doi: 10.1007/978-3-642-37036-6_13. URL `https://doi.org/10.1007/978-3-642-37036-6_13`.

Markus von Detten, Christian Heinzemann, Marie Christian Platenius, Jan Rieke, and Dietrich Travkin. Story diagrams - syntax and semantics. Technical report, University of Paderborn, 2012.

Dennis Wagelaar, Jörn Guy Süss, Hugo Bruneliere, Ronan, William Piers, Salva Martínez, and Thierry Fortin. Atl/user guide - the atl language, September 2014. URL `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`.

Junhua Wang, Soon-Kyeong Kim, and David A. Carrington. Verifying metamodel coverage of model transformations. In *17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia*, pages 270–282, 2006. doi: 10.1109/ASWEC.2006.55. URL `http://dx.doi.org/10.1109/ASWEC.2006.55`.

Xiaoliang Wang, Fabian Büttner, and Yngve Lamo. Verification of graph-based model transformations using alloy. *ECEASST*, 67, 2014. URL `http://journal.ub.tu-berlin.de/eceasst/article/view/943`.

Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, October 1998. ISBN 0201379406. URL `http://www.worldcat.org/isbn/0201379406`.

Glynn Winskel. *The formal semantics of programming languages - an introduction*. MIT Press, 1993a. ISBN 978-0-262-23169-5.

Glynn Winskel. *Information Systems*, chapter 12. In Winskel [1993a], 1993b. ISBN 978-0-262-23169-5.

Niklaus Wirth. *Compiler Construction*. International computer science series. Addison-Wesley, 1996. ISBN 9780201403534.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012. ISBN 978-3-642-29043-5. doi: 10.1007/978-3-642-29044-2. URL `http://dx.doi.org/10.1007/978-3-642-29044-2`.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 249–257, 1998. doi: 10.1145/277650.277732. URL `http://doi.acm.org/10.1145/277650.277732`.

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 164–173, 2007. doi: 10.1145/1321631.1321657. URL `http://doi.acm.org/10.1145/1321631.1321657`.

Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 315–324, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595757. URL `http://doi.acm.org/10.1145/1595696.1595757`.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation,* *PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294, 2011. doi: 10.1145/1993498.1993532. URL `http://doi.acm.org/10.1145/1993498.1993532`.

Appendix A

Modernization Transformation Proofs

Proposition 4.1. *For each concrete execution path* $\pi = (\sigma_{\text{in}}, \sigma_1, \ldots \sigma_{\text{out}})$ *of the program P, there exists the corresponding symbolic execution path* $\widehat{\pi} = ((\widehat{\sigma}_{\text{in}}, \text{true}), \ldots (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}))$*, such that the following equations are satisfied:*

$$\begin{aligned}
\widehat{\sigma}_{\text{in}} &= [x_i \mapsto x_1^?, \ldots, x_{\text{n}} \mapsto x_{\text{n}}^?] \\
\theta &= [x_0^? \mapsto \sigma_{\text{in}}(x_0), \ldots, x_{\text{n}}^? \mapsto \sigma_{\text{in}}(x_{\text{n}})] \\
((\widehat{\sigma}_{\text{out}})\theta)(\text{ret}) &= \sigma_{\text{out}}(\text{ret}) \\
(\widehat{b})\theta &= \text{true}
\end{aligned}$$

Proof. By induction on the length of the path π, and by showing that each concrete transition $\sigma_i \xrightarrow{s} \sigma_{i+1}$ from π can be simulated symbolically with some transition $(\widehat{\sigma}_i, \widehat{b}_i) \xrightarrow{s} (\widehat{\sigma}_{i+1}, \widehat{b}_{i+1})$ in a corresponding $\widehat{\pi}$. Let θ be a substitution assigning each symbol $\widehat{x}_i$ the value $\sigma_0(\widehat{x}_i)$. We say that a constrained symbolic store $(\widehat{\sigma}, \widehat{b})$ simulates a concrete store σ if $\sigma = (\widehat{\sigma})\theta$ and $(\widehat{b})\theta$ is true. Now, let $\sigma_i \xrightarrow{s} \sigma_{i+1}$ be a transition in π and σ_i be simulated by $(\widehat{\sigma}_i, \widehat{b}_i)$; this transition can be symbolically executed resulting in $(\widehat{\sigma}_i, \widehat{b}_i) \xrightarrow{s} (\widehat{\sigma}_i, \widehat{b}_{i+1})$ so σ_{i+1} is simulated by $(\widehat{\sigma}_{i+1}, \widehat{b}_{i+1})$. This way, by simulating each transition from π symbolically, we can generate the corresponding symbolic path $\widehat{\pi}$ fulfilling the stated requirements. $\square$

Theorem 4.1. *Two programs P and P′ are semantically equivalent* $P \sim P'$ *iff for each value* $v \in \text{Val}$ *it holds:*

$$\bigvee_{(\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}) \in \Sigma} (\widehat{\sigma}_{\text{out}}(\text{ret}) = v \wedge \widehat{b}_{\text{out}}) \iff \bigvee_{(\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}}) \in \Sigma'} (\widehat{\sigma}'_{\text{out}}(\text{ret}) = v \wedge \widehat{b}'_{\text{out}})$$

where

$$\Sigma = \left\{ (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}}) \;\middle|\; ((\widehat{\sigma}_{\text{in}}, \text{true}) \ldots (\widehat{\sigma}_{\text{out}}, \widehat{b}_{\text{out}})) \textit{ is a symbolic path of } P \right\}$$
$$\Sigma' = \left\{ (\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}}) \;\middle|\; ((\widehat{\sigma}_{\text{in}}, \text{true}) \ldots (\widehat{\sigma}'_{\text{out}}, \widehat{b}'_{\text{out}})) \textit{ is a symbolic path of } P' \right\}$$

Proof. Follows directly from Proposition 4.1 and Definition 4.6. □

Appendix B

SymexTRON Auxiliary Functions

$$\widehat{\text{singleton}}(\widehat{e},\widehat{h}) = \begin{cases}\{(x^?,\widehat{h})\} & \text{if } \widehat{e} = \{x^?\}\\ \widehat{\text{mk-singleton}}(\widehat{e},\widehat{h}) & \text{otherwise}\end{cases}$$

$$\widehat{\text{mk-singleton}}(\widehat{e},(\widehat{z},\widehat{\ell},\widehat{d},\widehat{\Gamma},\widehat{b})) = \left\{(x^?,\widehat{h}') \;\middle|\; x^? \text{ fresh} \wedge \widehat{h}' \text{ sat}\right\}$$

$$\begin{aligned}&\text{where } x^? \text{ fresh}\\ &\qquad \widehat{\Gamma}' = \widehat{\Gamma}[x^? \mapsto \widehat{\text{types}}(\widehat{e},\widehat{\Gamma},\widehat{z})]\\ &\qquad \widehat{h}' = (\widehat{z},\widehat{\ell},\widehat{d},\widehat{\Gamma}',\widehat{b} \wedge \widehat{e} = \{x^?\})\end{aligned}$$

$$\widehat{\text{inst}}(x^?,\widehat{h}) = \begin{cases}\{(\widehat{z}(x^?),\widehat{h})\} & \text{if } x^? \in \operatorname{dom}\widehat{z}\\ \widehat{\text{mk-inst}}(x^?,\widehat{h}) \cup \widehat{\text{alias-inst}}(x^?,\widehat{h}) & \text{otherwise}\end{cases}$$

$$\text{where } \widehat{h} = (\widehat{z},\widehat{\ell},\widehat{d},\widehat{\Gamma},\widehat{b})$$

$$\widehat{\text{mk-inst}}(x^?,\widehat{h}) = \left\{(\widehat{o},\widehat{h}') \;\middle|\; \begin{array}{l} c \in cs_{\text{in}} \wedge \\ \quad \widehat{h}' = \widehat{\text{mk-inst-typ}}(x^?,\widehat{o},c,cs_{\text{ex}},\widehat{h}) \wedge \widehat{h}' \text{ sat}\end{array}\right\}$$

$$\begin{aligned}&\text{where } \widehat{o} \text{ fresh}\\ &\qquad \widehat{h} = (\widehat{z},\widehat{\ell},\widehat{d},\widehat{\Gamma},\widehat{b})\\ &\qquad (cs_{\text{in}},cs_{\text{ex}}) = \widehat{\Gamma}(x^?)\end{aligned}$$

$$\widehat{\text{mk-inst-typ}}(x^?,\widehat{o},c,cs_{\text{ex}},\widehat{h}) = (\widehat{z}[x^? \mapsto \widehat{o}],\widehat{\ell}',\widehat{d},\widehat{\Gamma}',\widehat{b})$$

$$\begin{aligned}&\text{where } (\widehat{fs},\Gamma'') = \widehat{\text{mk-fields}}(\text{fields}(c),\widehat{\Gamma})\\ &\qquad \widehat{\ell}' = \widehat{\ell}\,[(\widehat{o},f) \mapsto \widehat{e} | (f,\widehat{e}) \in \widehat{fs}]\\ &\qquad \widehat{\Gamma}' = \widehat{\Gamma}''\left[\widehat{o} \mapsto (\{c\},\left\{c' \;\middle|\; \begin{array}{c} c' \mathsf{\ gen}^* \ c \wedge \\ c' \in cs_{\text{ex}}\end{array}\right\})\right]\end{aligned}$$

$$\widehat{\text{mk-fields}}(fs_c, \widehat{\Gamma}) = \widehat{\text{mk-fields}}'(\emptyset, fs_c, \widehat{\Gamma})$$

$$\widehat{\text{mk-fields}}'(fs_e, fs_c, \widehat{\Gamma}) = \begin{cases} \widehat{\text{mk-fields}}'(fs_e \cup \{(f, X^?)\}, fs'_c, \widehat{\Gamma}') & \text{if } fs_c = (f, c) \uplus fs'_c \\ (fs_e, \widehat{\Gamma}) & \text{otherwise} \end{cases}$$

$$\text{where } X^? \text{ fresh}$$
$$\widehat{\Gamma}' = \widehat{\Gamma}[X^? \mapsto c]$$

$$\widehat{\text{alias-inst}}(x^?, \widehat{h}) = \left\{ (\widehat{o}, \widehat{h}') \middle| \begin{array}{l} \widehat{o} \in \widehat{\text{type-insts}}(\widehat{cs}_{\text{in}}, \widehat{\Gamma}) \land \\ \quad \widehat{h}' = \widehat{\text{alias-inst-agn}}(x^?, \widehat{o}, cs_{\text{out}}, \widehat{h}) \ \land \widehat{h}' \text{ sat} \end{array} \right\}$$
$$\text{where } (cs_{\text{in}}, cs_{\text{out}}) = \Gamma(x^?)$$

$$\widehat{\text{alias-inst-agn}}(x^?, \widehat{o}, cs_{\text{out}}, \widehat{h}) = (\widehat{z}[x^? \mapsto \widehat{o}], \widehat{\ell}, \widehat{d}, \widehat{\Gamma}', \widehat{b})$$
$$\text{where } (\{c'\}, cs''_{\text{out}}) = \widehat{\Gamma}(o^?)$$
$$cs'_{\text{out}} = cs''_{\text{out}} \cup \{c | c \in cs_{\text{out}} \ \land c \text{ gen}^* \ c'\}$$
$$\widehat{\Gamma}' = \widehat{\Gamma}[\widehat{o} \mapsto (\{c'\}, cs'_{\text{out}})] \setminus \widehat{x}$$

$$\widehat{\text{type-insts}}(cs, \widehat{\Gamma}) = \left\{ \widehat{o} \ \middle| \ \begin{array}{l} (\{c'\}, cs_{\text{out}}) = \widehat{\Gamma}(o) \land \\ \quad (\exists c \in cs. c \text{ gen}^* \ c' \land \nexists c'' \in cs_{\text{out}}. c \text{ gen}^* \ c'') \end{array} \right\}$$

$$\widehat{\text{dc-containment}}(\widehat{e}, c, \widehat{z}, \widehat{d}, \widehat{\Gamma}) = \widehat{\text{dc-reown}}(\widehat{e}, c, \widehat{dcs}', \emptyset, \widehat{z}, \widehat{\Gamma}, \textbf{true})$$
$$\text{where } \widehat{dcs}' = \left\{ (\widehat{o}, c', \widehat{e}' \setminus \widehat{e}) \middle| (\widehat{o}, c', \widehat{e}') \in \text{graph}\, \widehat{d} \right\}$$

$$\widehat{\text{dc-reown}}(\widehat{e}, c, \widehat{dcs}, \widehat{dcs}'', \widehat{z}, \widehat{\Gamma}, \widehat{b}) = \begin{cases} \widehat{\text{dc-reown}}(\widehat{e}, c, dcs', \widehat{dcs}'' \cup \{\widehat{dc}'\}, \Gamma', \widehat{b} \land \widehat{b}') & \text{if } \widehat{dcs} = \widehat{dc} \uplus dcs' \\ \left[(\widehat{o}, c') \mapsto \widehat{e}'' \middle| (\widehat{o}, c', \widehat{e}'') \in \widehat{dcs}'' \right] & \text{otherwise} \end{cases}$$
$$\text{where } (\widehat{dc}', \widehat{\Gamma}', \widehat{b}') = \widehat{\text{dc-reown-1}}(\widehat{e}, \widehat{c}, \widehat{dc}, \widehat{z}, \widehat{\Gamma})$$

$$\widehat{\text{dc-reown-1}}(\widehat{e}, c, \widehat{dc}, \widehat{z}, \widehat{\Gamma}) = \begin{cases} \widehat{\text{dc-reown-1-sub}}(\widehat{e}, c, \widehat{dc}, \widehat{\Gamma}) & \text{if } c <: c' \\ \widehat{\text{dc-reown-1-sup}}(\widehat{e}, c, \widehat{dc}, \widehat{z}, \widehat{\Gamma}) & \text{if } c' <: c \land c \neq c' \\ (\widehat{dc}, \widehat{\Gamma}, \textbf{true}) & \text{otherwise} \end{cases}$$

$$\widehat{\text{dc-reown-1-sub}}(\widehat{e}, c, (\widehat{o}, c', \widehat{e}'), \widehat{\Gamma}) = ((\widehat{o}, c', \widehat{e}' \cup X^?), \widehat{\Gamma}[X^? \mapsto c], X^? = \emptyset \lor X^? = \widehat{e})$$
$$\text{where } X^? \text{ fresh}$$

$$\widehat{\text{dc-reown-1-sup}}(\widehat{e}, c, (\widehat{o}, c', \widehat{e}'), \widehat{z}, \widehat{\Gamma}) = ((\widehat{o}, c', \widehat{e}' \cup X^?), \widehat{\Gamma}', \widehat{b})$$

$$\begin{aligned}
&\text{where } X^?, Y^?, Z^? \text{ fresh}\\
&\qquad (cs_{\text{in}}, cs_{\text{ex}}) = \widehat{\text{types}}(\widehat{e}, \widehat{\Gamma}, \widehat{z})\\
&\qquad \widehat{\Gamma}' = \widehat{\Gamma}[X^? \mapsto c, Y^? \mapsto c', Z^? \mapsto (cs_{\text{in}}, cs_{\text{ex}} \cup c')]\\
&\qquad \widehat{b} = (X^? = \emptyset \vee X^? = Y^?) \wedge \widehat{e} = Y^? \uplus Z^?
\end{aligned}$$

$$\widehat{\text{match}}(x^?, X^?, c, \widehat{h}) = \widehat{\text{match-sucs}}(x^?, c, \widehat{h}) \cup \widehat{\text{match-fail}}(x^?, X^?, c, \widehat{h})$$

$$\widehat{\text{match-sucs}}(x^?, c, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b})) = \left\{ (\text{tt}, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}', \widehat{b})) \,\middle|\, \begin{array}{c} (\exists c' \in cs_{\text{in}}.c' <: c \vee c <: c') \wedge \\ (\nexists c' \in cs_{\text{ex}}.c' <: c) \end{array} \right\}$$

$$\begin{aligned}
&\text{where } (cs_{\text{in}}, cs_{\text{ex}}) = \widehat{\text{typeof}}(x^?, \widehat{\Gamma}, \widehat{z})\\
&\qquad \widehat{\Gamma}' = \begin{cases} \widehat{\Gamma}[\widehat{z}(x^?) \mapsto (c, \{c' | c' \in cs_{\text{ex}} \wedge c' <: c\})] & \text{if } x^? \in \operatorname{dom} \widehat{z} \\ \widehat{\Gamma}[x^? \mapsto (c, \{c' | c' \in cs_{\text{ex}} \wedge c' <: c\})] & \text{otherwise} \end{cases}
\end{aligned}$$

$$\widehat{\text{match-fail}}(x^?, X^?, c, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b})) = \left\{ (\text{ff}, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}', \widehat{b})) \,\middle|\, \begin{array}{c} (\exists c' \in cs_{\text{in}}.c \not<: c') \vee \\ (\exists c' \in cs_{\text{ex}}.c' <: c) \end{array} \right\}$$

$$\begin{aligned}
&\text{where } (cs_{\text{in}}, cs_{\text{ex}}) = \widehat{\text{typeof}}(x^?, \widehat{\Gamma}, \widehat{z})\\
&\qquad (cs'_{\text{in}}, cs'_{\text{ex}}) = \Gamma(X^?)\\
&\qquad \widehat{\Gamma}' = \begin{cases} \Gamma[\widehat{z}(x^?) \mapsto (cs_{\text{in}}, cs_{\text{ex}} \cup c), X^? \mapsto (cs'_{\text{in}}, cs'_{\text{ex}} \cup c)] & \text{if } x^? \in \operatorname{dom} \widehat{z} \\ \Gamma[x^? \mapsto (cs_{\text{in}}, cs_{\text{ex}} \cup c), X^? \mapsto (cs'_{\text{in}}, cs'_{\text{ex}} \cup c)] & \text{otherwise} \end{cases}
\end{aligned}$$

$$\widehat{\text{dcs}}(x^?, c, \widehat{d}, \widehat{h}) = \left\{ (\widehat{e}, \widehat{h}') \,\middle|\, (\widehat{o}, \widehat{h}'') \in \widehat{\text{inst}}(x^?, \widehat{h}) \wedge (\widehat{e}, \widehat{h}') = \widehat{\text{dcs}}'(\widehat{o}, c, \widehat{d}, \widehat{h}'') \right\}$$

$$\widehat{\text{dcs}}'(\widehat{o}, c, \widehat{d}_{\text{o}}, \widehat{h}) = \begin{cases} (\widehat{d}_{\text{o}}(\widehat{o}, c), \widehat{h}) & \text{if } (\widehat{o}, c) \in \operatorname{dom} \widehat{d}_{\text{o}} \\ (X^?, (\widehat{z}, \widehat{\ell}, \widehat{d}', \widehat{\Gamma}', \widehat{b})) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&\text{where } X^? \text{fresh}\\
&\qquad \widehat{h} = (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}, \widehat{b})\\
&\qquad \widehat{d}' = \widehat{d}[(\widehat{o}, c) \mapsto X^?]\\
&\qquad \widehat{\Gamma}' = \widehat{\Gamma}[X^? \mapsto c]
\end{aligned}$$

$$\begin{aligned}
\widehat{\text{types}}(X^?, \widehat{\Gamma}, \widehat{z}) &= \Gamma(X^?)\\
\widehat{\text{types}}(\emptyset, \widehat{\Gamma}, \widehat{z}) &= (\emptyset, \emptyset)\\
\widehat{\text{types}}(\{x^?_1, \ldots, x^?_n\}, \widehat{\Gamma}, \widehat{z}) &= \bigsqcup_{i=1}^{n} \widehat{\text{typeof}}(x^?_i, \Gamma, z)\\
\widehat{\text{types}}(\widehat{e}_1 \cup \widehat{e}_2, \widehat{\Gamma}, \widehat{z}) &= \widehat{\text{types}}(\widehat{e}_1, \widehat{\Gamma}, \widehat{z}) \sqcup \widehat{\text{types}}(\widehat{e}_2, \widehat{\Gamma}, \widehat{z})\\
\widehat{\text{types}}(\widehat{e}_1 \cap \widehat{e}_2, \widehat{\Gamma}, \widehat{z}) &= \widehat{\text{types}}(\widehat{e}_1, \widehat{\Gamma}, \widehat{z}) \sqcup \widehat{\text{types}}(\widehat{e}_2, \widehat{\Gamma}, \widehat{z})\\
\widehat{\text{types}}(\widehat{e}_1 \setminus \widehat{e}_2, \widehat{\Gamma}, \widehat{z}) &= \widehat{\text{types}}(\widehat{e}_1, \widehat{\Gamma}, \widehat{z})
\end{aligned}$$

$$\widehat{\text{typeof}}(x^?, \widehat{\Gamma}, \widehat{z}) = \begin{cases} \widehat{\Gamma}(\widehat{z}(x^?)) & \text{if } x^? \in \operatorname{dom} \widehat{z} \\ \widehat{\Gamma}(x^?) & \text{otherwise} \end{cases}$$

$$\begin{aligned} (cs_{\text{in}}, cs_{\text{ex}}) \sqcup (cs'_{\text{in}}, cs'_{\text{ex}}) &= (\widehat{\text{type-ub}}(cs_{\text{in}}, cs'_{\text{in}}), \widehat{\text{type-lb}}(cs_{\text{ex}}, cs'_{\text{ex}})) \\ \text{where } \widehat{\text{type-ub}}(cs, cs') &= \{c \mid c \in cs \cup cs' \wedge \nexists c' \in cs \cup cs'.c <: c'\} \\ \widehat{\text{type-lb}}(cs, cs') &= \{c \mid c \in cs \cap cs' \wedge \nexists c' \in cs \cap cs'.c <: c'\} \end{aligned}$$

$$\begin{aligned} &\widehat{\text{partition}}(\widehat{e}, (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}', \widehat{b})) = (\widehat{z}, \widehat{\ell}, \widehat{d}, \widehat{\Gamma}', \widehat{b} \wedge \widehat{e} = \{x^?\} \uplus X^?) \\ &\quad \text{where } x^?, X^? \text{ fresh} \\ &\qquad (cs_{\text{in}}, cs_{\text{ex}}) = \widehat{\text{types}}(\widehat{e}, \widehat{\Gamma}, \widehat{z}) \\ &\qquad \widehat{\Gamma}' = \widehat{\Gamma}[x^? \mapsto (cs_{\text{in}}, cs_{\text{ex}}), X^? \mapsto (cs_{\text{in}}, cs_{\text{ex}})] \end{aligned}$$

Appendix C

SymexTRON Proofs

Theorem 5.1 (Soundness). *If* $\exists m.\sigma = m(\widehat{\sigma}) \wedge \Gamma, h \overset{m}{\models} \widehat{h}$ *and* $\widehat{\mathcal{S}}[\![s]\!](\widehat{\sigma}, \widehat{h}) = \widehat{\mathbf{M}}$ *then for all* $(\widehat{\sigma}, \widehat{h}) \in \widehat{\mathbf{M}}$ *there exists* σ', Γ', *and* h' *such that we have a concrete execution* $s, \sigma, \Gamma, h \Longrightarrow \sigma', \Gamma', h'$ *and exists a model* m' *such that* $\sigma' = m'(\widehat{\sigma}')$ *and* $\Gamma', h' \overset{m'}{\models} \widehat{h}'$.

Proof Sketch. The basic rules follow standard techniques from General Symbolic Execution Khurshid et al. [2003] and the Lazier# algorithm Deng et al. [2012]. The only non-standard components of our formal algorithm are type bounds, containment updates, lazy iteration and deep containment constraints. We discuss the soundness of these components:

- Type bounds are precise for symbolic instances, but may over-approximate for symbols and symbolic references. In particular, during partitioning of symbolic expressions, new symbols and symbolic reference sets are created and their assigned types is inferred from the symbolic expression. Despite this local over-approximation soundness is still preserved, since partitioning constraints are never dropped globally and we can thus only assign symbolic instances to symbols—which is an under-approximative operation—if the assignment agrees with these global constraints (otherwise the resulting heap is unsatisfiable and not considered).

- The containment property, means that each instance is at most contained in a single other instance, and that containment is acyclic. To ensure that this property is preserved in concrete field updates, the rule checks acyclicity before rule assignment and then proceeds to remove the set of newly assigned instances from the other containment fields. Both acyclicity and unique ownership are correspondingly ensured by the symbolic semantics. The acyclicity constraint is encoded in the symbolic constraints of containment, and thus cyclic assignment produces an unsatisfiable heap. The unique ownership constraint is ensured by the $\widehat{\text{disown}}$ function which instead uses the set difference operation to ensure that the assigned symbolic set references do not contain instances symbolically represented by the newly assigned symbolic expression; since containment constraints are also affected, it ensures to update these as well with the dc-$\widehat{\text{ownership}}$ function.

- Lazy iteration should model concrete iteration and so should symbolically represent the set of values matched in a particular state. However, because of laziness of the operation and possibility of state change at each iteration it is required that the $\widehat{\text{next}}$ operation keeps track of the original set of descendant constraints when performing a new iteration, since otherwise the heap can become inconsistent with concrete iteration.
- Containment constraints are additionally to being consistently updated as mentioned above, also encoded directly as a constraint in the model finder for the symbolic heap. This means that it is only possible to make assignments for referenced symbols to symbolic instances in a way consistent with the type and containment constraints; any non-sound assignment will produce an unsatisfiable symbolic heap, which is not further considered in the execution.

$\square$

Appendix D

TRON Subject programs

Data model for RenameField, RenameMethod, ExtractSuper, ReplaceDelegation:

```
class Package {
  classes :◆ Class*
}
class Class {
  name    :  String
  super   :⇝ Class
  fields  :◆ Field*
  methods :◆ Method*
}
class Field {
  name :  String
  type :⇝ Class
}
class Method {
  name   :  String
  params :◆ Parameter*
  body   :◆ Statement
  type   :⇝ Class
}
class Parameter {
  name :  String
  type :⇝ Class
}
class Statement {}
class IfStatement extends Statement {
  then :◆ Statement
  else :◆ Statement
  cond :◆ Expr
}
class Return extends Statement {
  value :◆ Expr
}
class Assign Statement {
  left  :◆ AssignableExpr
  right :◆ Expr
}
class Expr {
  type :⇝ Class
}
class AssignableExpr extends Expr { }
class FieldAccessExpr extends AssignableExpr {
  field_name :   String
  target     :◆ Expr
}
class MethodCallExpr extends Expr {
  method_name :   String
  target      :◆ Expr
  args        :◆ Arg*
}
class Arg {
  name  :   String
  value :◆ Expr
}
```

RenameField:

```
class_fields := class.fields;
class.fields := (class_fields \ old_field) ∪ new_field;
foreach faexpr ∈ package match* FieldAccessExpr do
  faexpr_field_name := faexpr.field_name;
  old_field_name := old_field.name;
  faexpr_target := faexpr.target;
  faexpr_target_type := faexpr_target.type;
  if faexpr_field_name = old_field_name ∧
             class = faexpr_target_type then
    new_field_name := new_field.name;
    faexpr.field_name := new_field_name
  else skip
```

RenameMethod:

```
class_methods := class.methods;
class.methods := (class_methods \ old_method) ∪ new_method;
foreach mcexpr ∈ package match* MethodCallExpr do
  mcexpr_method_name := mcexpr.method_name;
  old_method_name := old_method.name;
  old_method_params := old_method.params;
  mcexpr_target := mcexpr.target;
  mcexpr_target_type := mcexpr_target.type;
  mcexpr_args := mcexpr.args;
  paramsmatched := new Any;
  foreach omp ∈ old_method_params do
    parammatched := ∅;
    omp_name := omp.name;
    foreach mcea ∈ mcexpr_args do
      mcea_name := mcea.name;
      if omp_name = mcea_name then
        parammatched := paramsmatched
      else skip;
    if parammatched = ∅ then
      paramsmatched := ∅
    else skip;
  if mcexpr_method_name = old_method_name $\wedge$
        class = mcexpr_target_type $\wedge$
          (($\neg$(old_method_params = $\emptyset$) $\wedge$ mcexpr_args = $\emptyset$)
             $\vee$ paramsmatched = $\emptyset$)
  then
    new_method_name := new_method.name;
    mcexpr.method_name := new_method_name
  else skip
```

ExtractSuper:

```
sclass := new Class;
package_classes := package.classes;
package.classes := package_classes $\cup$ sclass;
class1.super := sclass;
class2.super := sclass;
sclass.name := sc_name;
new_sclass_fields := $\emptyset$;
rem_class1_fields := $\emptyset$;
rem_class2_fields := $\emptyset$;
class1_fields := class1.fields;
class2_fields := class2.fields;
foreach c1f $\in$ class1_fields do
  foreach c2f $\in$ class2_fields do
    c1f_name := c1f.name;
    c2f_name := c2f.name;
    c1f_type := c1f.type;
    c2f_type := c2f.type;
    if c1f_name = c2f_name $\wedge$ c2f_type = c2f_type then
      scf := new Field;
      scf.name := c1f_name;
      scf.type := c1f_type;
      new_sclass_fields := new_sclass_fields $\cup$ scf;
      rem_class1_fields := rem_class1_fields $\cup$ c1f;
      rem_class2_fields := rem_class2_fields $\cup$ c2f
    else
      skip;
class1.fields := class1_fields $\setminus$ rem_class1_fields;
class2.fields := class2_fields $\setminus$ rem_class2_fields;
sclass.fields := new_sclass_fields
```

ReplaceDelegation:

```
class_fields := class.fields;
field_type := field.type;
class.super := field_type;
field_type_methods := field_type.methods;
class_methods := class.methods;
class_new_methods := $\emptyset$;
foreach ftm $\in$ field_type_methods do
  foreach cm $\in$ class_methods do
    ftm_name := ftm.name;
    cm_name := cm.name;
    if $\neg$(ftm_name = cm_name) then
      class_new_methods := class_new_methods $\cup$ cm
    else skip;
```

```
class.methods := class_new_methods;
foreach mcexpr ∈ class match⋆ MethodCallExpr do
  mcexpr_target := mcexpr.target;
  MCEXPR_TARGET := ∅;
  foreach mcx in mcexpr_target match FieldAccessExpr do
    MCEXPR_TARGET := mcx;
  if ¬(MCEXPR_TARGET = ∅) then
    mcexpr_target_target := mcexpr_target.target;
    mcexpr_target_target_type := mcexpr_target_target.type;
    mcexpr_target_field_name := mcexpr_target.field_name;
    field_name := field.name;
    if field_name = mcexpr_target_field_name ∧
          class = mcexpr_target_target_type then
      mcexpr.target := mcexpr_target_target
    else skip
  else skip;
class.fields := class_fields \ field
```

Data models for Fam2Pers:

```
// Families meta model
class Family {
  lastName  :   String
  father    :◆  Member  opposite familyFather
  mother    :◆  Member  opposite familyMother
  sons      :◆  Member⋆ opposite familySon
  daughters :◆  Member⋆ opposite familyDaughter
}
class Member {
  firstName      :   String
  familyFather   :⇝  Family? opposite father
  familyMother   :⇝  Family? opposite mother
  familySon      :⇝  Family? opposite sons
  familyDaughter :⇝  Family? opposite daughters
}
// Persons meta model
class Person {
  fullName : String
}
class Male extends Person { }
class Female extends Person { }
```

Fam2Pers:

```
persons := ∅;
foreach member ∈ families match⋆ Member do
  self_familyMother := member.familyMother;
  self_familyDaughter := member.familyDaughter;
  // Start inlined isFemale helper
  if ¬(self_familyMother = ∅) then
    isFemale := new Any
  else if ¬(self_familyDaughter = ∅) then
    isFemale := new Any
  else
    isFemale := ∅;
  // End inlined isFemale helper
  if ¬(isFemale = ∅) then
    female := new Female;
    self_familyFather := member.familyFather;
    self_familyMother := member.familyMother;
    self_familySon := member.familySon;
    self_familyDaughter := member.familyDaughter;
    // Start inlined familyName helper
    if ¬(self_familyFather = ∅) then
      familyName := self_familyFather.lastName
    else if ¬(self_familyMother = ∅) then
      familyName := self_familyMother.lastName
    else if ¬(self_familySon = ∅) then
      familyName := self_familySon.lastName
    else
      familyName := self_familyDaughter.lastName;
    // End inlined familyName helper
    member_firstName := member.firstName;
    fullName := new Concat;
    fullName.s1 := member_firstName;
    fullName.s2 := familyName;
    female.fullName := fullName;
    persons := persons ∪ female
  else
    male := new Male;
    self_familyFather := member.familyFather;
    self_familyMother := member.familyMother;
    self_familySon := member.familySon;
    self_familyDaughter := member.familyDaughter;
    // Start inlined familyName helper
    if ¬(self_familyFather = ∅) then
      familyName := self_familyFather.lastName
    else if ¬(self_familyMother = ∅) then
      familyName := self_familyMother.lastName
    else if ¬(self_familySon = ∅) then
      familyName := self_familySon.lastName
    else
      familyName := self_familyDaughter.lastName;
    // End inlined familyName helper
    member_firstName := member.firstName;
    fullName := new Concat;
    fullName.s1 := member_firstName;
    fullName.s2 := familyName;
    male.fullName := fullName;
    persons := persons ∪ male
```

Data model for Path2Petri:

```
// Shared
class Element {
  name : String
}
// Path expression meta model
class PathExp extends Element {
  transitions :◆ PETransition⋆
  states      :◆ State⋆
}
class State extends Element {
  outgoing :⇝ PETransition* opposite source
  incoming :⇝ PETransition* opposite target
}
class PETransition extends Element {
  source   :⇝ State opposite outgoing
  target   :⇝ State opposite incoming
}
// Petri net meta model
class PetriNet extends Element {
  transitions :◆ PNTransition*
  arcs        :◆ Arc*
  place       :◆ Place*
}
class PNTransition extends Element {
  outgoing :⇝ TransToPlaceArc* opposite source
  incoming :⇝ PlaceToTransArc* opposite target
}
class Place extends Element {
  outgoing :⇝ PlaceToTransArc* opposite source
  incoming :⇝ TransToPlaceArc* opposite target
}
class Arc extends Element {
  weight : Integer
}
class TransToPlaceArc extends Arc {
  source :⇝ PNTransition opposite outgoing
  target :⇝ Place opposite incoming
}
class PlaceToTransArc extends Arc {
  source :⇝ Place opposite incoming
  target :⇝ PNTransition opposite outgoing
}
```

Path2Petri:

```
places := ∅;
transitions := ∅;
eString := new Empty;
int1 := new Int;
// First pass to create places
foreach st ∈ pe match* State do
  place := new Place;
  st._Place := place;
  place.name := eString;
  places := places ∪ place;
foreach tr ∈ pe match* PETransition do
  pntr := new PNTransition;
  tr._PNTransition := pntr;
  tr_name := tr.name;
  pntr.name := tr_name;
  pnia := new PlaceToTransArc;
  tr._PN_IA := pnia;
  pntr.incoming := pnia;
  tr_source := tr.source;
  tr_source_Place := tr_source._Place;
  pnia.source := tr_source_Place;
  pnia.target := pntr;
  pnia.weight := int1;
  pnoa := new TransToPlaceArc;
  tr._PN_OA := pnoa;
  pntr.outgoing := pnoa;
  pnoa.source := pntr;
  tr_target := tr.target;
  tr_target_Place := tr_target._Place;
  pnoa.target := tr_target_Place;
  pnia.weight := int1;
  transitions := transitions ∪ pntr;
// Second pass to link places to arcs
foreach st ∈ pe match* State do
  st_Place := st._Place;
  pnoas := ∅;
  st_incoming := st.incoming;
  foreach inc ∈ st_incoming do
    inc_PN_OA := inc._PN_OA;
    pnoas := pnoas ∪ inc_PN_OA;
  st_Place.incoming := pnoas;
  pnias := ∅;
  st_outgoing := st.outgoing;
  foreach outg ∈ st_outgoing do
    outg_PN_IA := outg._PN_IA;
    pnias := pnias ∪ outg_PN_IA;
  st_Place.outgoing := pnias;
pn := new PetriNet;
pe_name := pe.name;
pn.name := pe_name;
pn.places := places;
pn.transitions := transitions;
arcs := ∅;
foreach pntr ∈ transitions do
  pnia := pntr._PN_IA;
  pnoa := pntr._PN_OA;
  arcs := arcs ∪ pnia ∪ pnoa;
pn.arcs := arcs
```

Data models for Class2Rel:

```
// Class meta model
class NamedElt {
  name : String
}
class Package {
  classifiers :◆ Classifier*
}
class Classifier extends NamedElt { }
class DataType extends Classifier { }
class Class extends Classifier {
  isAbstract :   Boolean
  attributes :◆ Attribute* opposite owner
  super      :⇝ Class?
}
class Attribute extends NamedElt {
  isMultivalued :   Boolean
  type          :⇝ Classifier
  owner         :⇝ Class opposite attribute
}
// Relational meta model
class Named {
  name : String
}
class Schema {
  tables :◆ Table*
  types  :◆ Type*
}
class Table extends Named {
  columns :◆ Column*
  key     :⇝ Column
}
class Column extends Named {
  type    :⇝ Type
}
```

Class2Rel:

```
objectIdType := new Type;
objectIdType.name := integer_name;
schema := new Schema;
foreach dt ∈ package match* DataType do
  dt_name := dt.name;
  if dt_name = integer_name then
    dt._Type := objectIdType
  else
    type := new Type;
    dt._Type := type;
    type.name := dt_name;
    schema_types := schema.types;
    schema.types := schema_types ∪ type;
idString := new String;
objectIdString := new String;
foreach at ∈ package match* Attribute do
  at_type := at.type;
  at_isMultivalued := at.isMultivalued;
  foreach _ ∈ at_type match DataType do
    if at_isMultivalued = ∅ then
      at_name := at.name;
      at_type_Type := at_type._Type;
      column := new Column;
      column.name := at_name;
      column.type := at_type_Type;
      at._Column := column
    else
     at_owner := at.owner;
     at_owner_name := at_owner.name;
     at_name := at.name;
     at_type_Type := at_type._Type;
     tableName := new Concat;
     tableName.s1 := at_owner_name;
     tableName.s2 := at_name;
     idName := new Concat;
     idName.s1 := at_owner_name;
     idName.s2 := idString;
     id := new Column;
```

```
      id.name := idName;
      id.type := objectIdType;
      value := new Column;
      value.name := at_name;
      value.type := at_type_Type;
      table := new Table;
      table.name := tableName;
      table.key := id;
      table.columns := id ∪ value;
      schema_tables := schema.tables;
      schema.tables := schema_tables ∪ table;
  foreach _ ∈ at_type match Class do
    if at_isMultivalued = ∅ then
      at_name := at.name;
      column_name := new Concat;
      column_name.s1 := at_name;
      column_name.s2 := idString;
      column := new Column;
      column.name := column_name;
      column.type := objectIdType;
      at._Column := column
    else
      at_owner := at.owner;
      at_owner_name := at_owner.name;
      at_name := at.name;
      tableName := new Concat;
      tableName.s1 := at_owner_name;
      tableName.s2 := at_name;
      idName := new Concat;
      idName.s1 := at_owner_name;
      idName.s2 := idString;
      id := new Column;
      id.name := idName;
      id.type := objectIdType;
      foreignKey := new Column;
      foreignKey.name := at_name;
      foreignKey.type := objectIdType;
      table := new Table;
      table.name := tableName;
      table.key := id;
      table.columns := id ∪ foreignKey;
      schema_tables := schema.tables;
      schema.tables := schema_tables ∪ table;
  foreach class ∈ package match* Class do
    class_name := class.name;
    class_attributes := class.attributes;
    key := new Column;
    key.name := objectIdString;
    key.type := objectIdType;
    cols := key;
    foreach at ∈ class_attributes do
      at_isMultivalued := at.isMultivalued;
      if at_isMultivalued = ∅ then
        at_Column := at._Column;
        cols := cols ∪ at_Column
      else skip;
    table := new Table;
    table.name := class_name;
    table.key := key;
    table.columns := cols;
    schema_tables := schema.tables;
    schema.tables := schema_tables ∪ table
```

Toy1:

```
containselem := ∅;
foreach sublist ∈ list match* IntList do
  sublist_data := sublist.data;
  if elem = sublist_data then
    containselem := new Any
  else skip
```

Toy2:

```
if list = ∅ then
  res := new Any
else
  head := list.data;
  list_next := list.next;
  if list_next = ∅ then
    res := new Any
  else
    fix list_next do
      list_next_next := list_next.next;
      if list_next_next = ∅ then
        tail := list_next.data
      else
        list_next := list_next_next;
    if head = tail then
      res := new Any
    else
      res := ∅
```

Toy3:

```
table := new Table;
idcol := new IdColumn;
table.id := idcol;
table.columns := idcol;
class_attributes := class.attributes;
foreach attr ∈ class_attributes do
  col := new DataColumn;
  attrtype := attr.type;
  col.type := attrtype;
  tablecolumns := table.columns;
  table.columns := tablecolumns ∪ col
```

Toy4:

```
table := new Table;
idcol := new IdColumn;
table.id := idcol;table.columns := idcol;
foreach attr ∈ class match* Attribute do
```

```
  col := new DataColumn;
  attrtype := attr.type;
  col.type := attrtype;
  tablecolumns := table.columns;
  table.columns := tablecolumns ∪ col
```

Toy5:

```
timestamps := ∅;
foreach ts ∈ post match* Timestamp do
  timestamps := timestamps ∪ ts
```

Toy6:

```
foreach sp ∈ post match* SinglePost do
  sp_title := sp.title;
  sp_title_value := sp_title.value;
  new_sp_title := new CapitalisedTitle;
  new_sp_title.value := sp_title_value;
  sp.title := new_sp_title
```

Toy7:

```
invitationlist := ∅;
foreach person ∈ contactbook match* Person do
  isadult := ∅;
  person_age := person.age;
  person_name := person.name;
  foreach age ∈ person_age match Adult do
    isadult := new Any;
  if ¬(isadult = ∅) then
    invited := new Invited;
    invited.name := person_name;
    invitationlist := invitationlist ∪ invited
  else skip
```

Appendix E

Rascal Light Semantics Proofs

Theorem 6.1 (Backtracking purity). *If* $\underline{cs}; v; \sigma \overset{\mathcal{CS}}{\underset{\text{cases}}{\Longrightarrow}} \textbf{fail}; \sigma'$ *then* $\sigma' = \sigma$

Proof. By induction on the derivation $\mathcal{CS}$:

- Case $\mathcal{CS} = \textsc{ECS-Emp}\dfrac{}{\varepsilon; v; \sigma \underset{\text{cases}}{\Longrightarrow} \textbf{fail}; \sigma}$, so $\underline{cs} = \varepsilon$ and $\sigma' = \sigma$. Holds by definition.

- Case $\mathcal{CS} = \textsc{ECS-More-Fail}\dfrac{\sigma \vdash p \overset{?}{:=} v \overset{\mathcal{M}}{\underset{\text{match}}{\Longrightarrow}} \underline{\rho} \qquad \underline{\rho}; e; \sigma \overset{\mathcal{C}}{\underset{\text{case}}{\Longrightarrow}} \textbf{fail}; \sigma'' \qquad \underline{cs'}; v; \sigma \overset{\mathcal{CS}'}{\underset{\text{cases}}{\Longrightarrow}} \textbf{fail}; \sigma'}{\textbf{case}\ p \Rightarrow e, \underline{cs'}; v; \sigma \underset{\text{cases}}{\Longrightarrow} \textbf{fail}; \sigma'}$, so $\underline{cs} = (\textbf{case}\ p \Rightarrow e, \underline{cs'})$ and $vres = \textbf{fail}$.
  - By inductive hypothesis of $\mathcal{CS}'$ we get $\sigma' = \sigma$.

- Note: the rule ECS-More-Ord is inapplicable since its premise states that the result value $vres \neq \textbf{fail}$.

□

In order to prove Theorem 6.2, we need to state some helper lemmas about sub-derivations.

We have a lemma for the auxiliary merge function:

Lemma E.1. *If we have* $\underline{\rho'} = \text{merge}(\underline{\rho_1}, \ldots, \underline{\rho_n})$ *and for each value* $v \in \text{img}\ \rho_{i,j}$ *in an environment in the input sequence of environment sequences* $\underline{\rho_1}, \ldots, \underline{\rho_n}$ *there exists a type* t *so that* $v : t$*, then we have that for each value* $v' \in \text{img}\ \rho'_k$ *in an environment in the resulting environment sequence* $\underline{\rho'}$ *we have a type* t' *so that* $v' : t'$

Proof. Follows directly from the premises by induction of the input sequence of environment sequences $\underline{\rho_1}, \ldots, \underline{\rho_n}$ □

We have a lemma for the auxiliary children function:

Lemma E.2. *For a value* v *such that we have* $\underline{v'} = children(v)$ *and* $\dfrac{\mathcal{T}}{v : t}$*, then there exists a type sequence* $\underline{t'}$ *such that* $\underline{v' : t'}$

Proof. By induction on the syntax of v:

- Cases $v = vb$ and $v = \blacksquare$, so $\underline{v'} = \varepsilon$. Holds trivially.
- Case $v = k(\underline{v'})$
  - By inversion we get

    $$\text{T-CONSTRUCTOR}\ \frac{\textbf{data}\ at = \ldots \mid k(\underline{t''}) \mid \ldots \qquad \underline{\dfrac{\mathcal{T}'}{v : t'}} \qquad \underline{\dfrac{\mathcal{ST}'}{t' <: t''}}}{k(\underline{v'}) : at}$$

    so $t = at$
  - Here, $\underline{\mathcal{T}'}$ exactly represents our target goal
- Case $v = [\underline{v'}]$
  - By inversion we get $\mathcal{T} = \text{T-LIST}\ \dfrac{\underline{\dfrac{\mathcal{T}'}{v' : t'}}}{[\underline{v'}] : \text{list}(\bigsqcup \underline{t'})}$, so $t = \text{list}(\bigsqcup \underline{t'})$
  - Here, $\underline{\mathcal{T}'}$ exactly represents our target goal
- Case $v = \{\underline{v'}\}$
  - By inversion we get $\mathcal{T} = \text{T-SET}\ \dfrac{\underline{\dfrac{\mathcal{T}'}{v' : t'}}}{\{\underline{v'}\} : \text{set}(\bigsqcup \underline{t'})}$, so $t = \text{set}(\bigsqcup \underline{t'})$
  - Here, $\underline{\mathcal{T}'}$ exactly represents our target goal

- Case $v = (\underline{v'' : v'''})$, so $\underline{v'} = \underline{v''}, \underline{v'''}$
  - By inversion we get

$$\mathcal{T} = \text{T-MAP}\dfrac{\overset{\mathcal{T}''}{\underline{v'' : t''}} \qquad \overset{\mathcal{T}'''}{\underline{v''' : t'''}}}{(\underline{v'' : v'''}) : \mathrm{map}(\bigsqcup \underline{t''}, \bigsqcup \underline{t'''})}$$

  - Here we take the concatenation of the premises $\mathcal{T}'', \mathcal{T}'''$ to fulfill our goal.

$\square$

We have two mutually recursive lemmas on pattern matching: Lemma E.3 and Lemma E.4.

Lemma E.3. *If* $\overset{\mathcal{M}}{\sigma \vdash p \overset{?}{:=} v \underset{match}{\Longrightarrow} \underline{\rho}}$ *and there exists a type t such that* $\overset{\mathcal{T}}{v : t}$ *and a type t' such that $v' : t'$ for each value in the input store $v' \in \mathrm{img}\ \sigma$, then we have a type t'' for each value $v'' \in \mathrm{img}\ \rho_i$ in an environment in the output sequence $\underline{\rho}$.*

Proof. By induction on the derivation $\mathcal{M}$:

- Cases $\mathcal{M}$ = P-VAL-SUCS, $\mathcal{M}$ = P-VAL-FAIL, $\mathcal{M}$ = P-VAR-UNI, $\mathcal{M}$ = P-VAR-FAIL, $\mathcal{M}$ = P-CONS-FAIL, $\mathcal{M}$ = P-TYPE-FAIL, $\mathcal{M}$ = P-LIST-FAIL, $\mathcal{M}$ = P-SET-FAIL, $\mathcal{M}$ = P-NEG-SUCS, $\mathcal{M}$ = P-NEG-FAIL hold trivially since we have that the output environment sequence $\underline{\rho} = []$ or $\underline{\rho} = \varepsilon$, in both cases containing no values.
- Case $\mathcal{M}$ = P-VAR-BIND holds by the premise derivation $\mathcal{T}$.
- Case $\mathcal{M} = \text{P-CONS-SUCS}\dfrac{\overset{\mathcal{M}'_1}{\sigma \vdash p'_1 \overset{?}{:=} v'_1 \underset{\text{match}}{\Longrightarrow} \underline{\rho'_1}} \quad \cdots \quad \overset{\mathcal{M}'_n}{\sigma \vdash p'_n \overset{?}{:=} v'_n \underset{\text{match}}{\Longrightarrow} \underline{\rho'_n}}}{\sigma \vdash k(\underline{p'}) \overset{?}{:=} k(\underline{v'}) \underset{\text{match}}{\Longrightarrow} \mathrm{merge}(\underline{\rho'_1}, \ldots, \underline{\rho'_n})}$,
  so $p = k(\underline{p'})$, $v = k(\underline{v'})$ and $\underline{\rho} = \mathrm{merge}(\underline{\rho'_1}, \ldots, \underline{\rho'_n})$
  - By inversion we get

$$\mathcal{T} = \text{T-CONSTRUCTOR}\dfrac{\textbf{data}\ at = \ldots \mid k(\underline{t''}) \mid \ldots \qquad \overset{\mathcal{T}'}{\underline{v : t'}} \qquad \overset{\mathcal{ST}'}{\underline{t' <: t''}}}{k(v'_1, \ldots, v'_n) : at}$$

  so $t = at$

  - Now by induction hypotheses of $\underline{\mathcal{M}'}$ using $\underline{\mathcal{T}'}$, and then using Lemma E.1 we get that $\underline{\rho}$ has well-typed values.

- Case $\mathcal{M} = \text{P-TYPE-SUCS}$ holds by the premise derivation $\mathcal{T}$ and Lemma E.1.

- Case $\mathcal{M} = \text{P-LIST-SUCS}\dfrac{\overset{\mathcal{MS}'}{\sigma \vdash \underline{\star p'} \overset{?}{:=} \underline{v'} \mid \emptyset \xRightarrow[\text{match}\star]{[]\ ,} \underline{\rho}}}{\sigma \vdash [\underline{\star p'}] \overset{?}{:=} [\underline{v'}] \xRightarrow[\text{match}]{} \underline{\rho}}$, so $p = [\underline{\star p'}]$ and $v = [\underline{v'}]$

  - By inversion we get $\mathcal{T} = \text{T-LIST}\dfrac{\overset{\mathcal{T}'}{\underline{v' : t'}}}{[\underline{v'}] : \text{list}(\bigsqcup \underline{t'})}$, so $t = \text{list}(\bigsqcup \underline{t'})$
  - Partitioning lists by splitting on concatenation , preserves typing since we can for each value pick the corresponding type derivation in the sequence.
  - By induction hypothesis (using Lemma E.4) of $\mathcal{MS}'$ using $\underline{\mathcal{T}'}$ and above fact we get that $\underline{\rho}$ has well-typed values.

- Case $\mathcal{M} = \text{P-SET-SUCS}\dfrac{\overset{\mathcal{MS}'}{\sigma \vdash \underline{\star p'} \overset{?}{:=} \underline{v'} \mid \emptyset \xRightarrow[\text{match}\star]{[]\ ,} \underline{\rho}}}{\sigma \vdash \{\underline{\star p'}\} \overset{?}{:=} \{\underline{v'}\} \xRightarrow[\text{match}]{} \underline{\rho}}$, so $p = \{\underline{\star p'}\}$ and $v = \{\underline{v'}\}$

  - By inversion we get $\mathcal{T} = \text{T-SET}\dfrac{\overset{\mathcal{T}'}{\underline{v' : t'}}}{\{\underline{v'}\} : \text{set}(\bigsqcup \underline{t'})}$, so $t = \text{set}(\bigsqcup \underline{t'})$
  - Partitioning sets using disjoint union $\uplus$ preserves typing of value sub-sequences, since we can for each value v_i in a subsequence pick the corresponding typing derivation $\mathcal{T}_i'$
  - By induction hypothesis (using Lemma E.4) of $\mathcal{MS}'$ using $\underline{\mathcal{T}'}$ and above fact we get that $\underline{\rho}$ has well-typed values.

- Case $\mathcal{M} = \textsc{P-Deep}\dfrac{\overset{\mathcal{M}'}{\sigma \vdash p' \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho'}} \qquad v'_1, \ldots, v'_n = \text{children}(v) \qquad \overset{\mathcal{M}''_1}{\sigma \vdash /p' \overset{?}{:=} v'_1 \xRightarrow[\text{match}]{} \underline{\rho''_1}} \quad \cdots \quad \overset{\mathcal{M}''_n}{\sigma \vdash /p' \overset{?}{:=} v'_n \xRightarrow[\text{match}]{} \underline{\rho''_n}}}{\sigma \vdash /p' \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho'}, \underline{\rho''_1}, \ldots, \underline{\rho''_n}}$
  - By induction hypothesis on $\mathcal{M}'$, we get that $\underline{\rho'}$ has well-typed values
  - By using Lemma E.2 on $\underline{v'}$, we get $\dfrac{\overset{\mathcal{T}'}{v' : t'}}{}$
  - By induction hypotheses on $\underline{\mathcal{M}''}$ using $\mathcal{T}'$ we get that $\underline{\rho''_1}, \ldots, \underline{\rho''_n}$ is well-typed
  - Now, we can show that $\underline{\rho'}, \underline{\rho''_1}, \ldots, \underline{\rho''_n}$ is well-typed using above facts

□

Lemma E.4. *If* $\overset{\mathcal{MS}}{\sigma \vdash \underline{\star p} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xRightarrow[\textit{match}\star]{() \quad \otimes} \underline{\rho}}$ *and the following properties hold:*

1. *There exists a type sequence* $\underline{t}$ *such that* $\underline{v : t}$
2. *There exists a type* t' *such that* $v' : t'$ *for each value in the input store* $v' \in \operatorname{img} \sigma$
3. *There exists a type* t' *such that* $v' : t'$ *for each value in the visited value set* $v' \in \mathbb{V}$
4. *If for all value sequences* $\underline{v'}, \underline{v''}, \underline{v'''}$ *where we have* $\underline{v'} = \underline{v''} \otimes \underline{v'''}$ *and there exists a type sequence* $\underline{t'}$ *so* $\overline{\underline{v' : t'}}$*, then we have that there exists two type sequences* $\underline{t''}$ *and* $\underline{t'''}$ *so that* $\underline{v'' : t''}$ *and* $\underline{v''' : t'''}$

Then we have a type t' *so that* $v' : t'$ *for each value* $v' \in \operatorname{img} \rho_i$ *in an environment in the output sequence* $\underline{\rho}$*.*

Proof. By induction on the derivation $\mathcal{MS}$:

- Cases $\mathcal{MS} = \textsc{PL-Emp-Both}$, $\mathcal{MS} = \textsc{PL-Emp-Pat}$, $\mathcal{MS} = \textsc{PL-Emp-Val}$, $\mathcal{MS} = \textsc{PL-More-Star-Pat-Fail}$, and $\mathcal{MS} = \textsc{PL-More-Star-Val-Fail}$ hold trivially since we have $\underline{\rho} = []$ or $\underline{\rho} = \varepsilon$, and there are no values in either $[]$ or ε.

- Case $\mathcal{MS} = \text{PL-M{\scriptsize ORE}-P{\scriptsize AT}}$ holds by induction hypotheses (including Lemma E.3), and Lemma E.1.

- Case $\mathcal{MS} = \text{PL-M{\scriptsize ORE}-S{\scriptsize TAR}-U{\scriptsize NI}}$ $$\frac{\begin{array}{c} x \in \text{dom}\,\sigma \qquad \sigma(x) = (\underline{v'}) \qquad \underline{v} = \underline{v'} \otimes \underline{v''} \\ \mathcal{MS'} \\ \sigma \vdash \underline{\star p'} \overset{?}{:=} \underline{v''} \mid \emptyset \xrightarrow[\text{match}\star]{() \quad \otimes} \underline{\rho} \end{array}}{\sigma \vdash \star x, \underline{\star p'} \overset{?}{:=} \underline{v} \mid \mathbb{V} \xrightarrow[\text{match}\star]{() \quad \otimes} \underline{\rho}},$$

  so $\underline{\star p} = \star x, \underline{\star p'}$.

  By property 4 we can derive that there exists a type sequence $\underline{t''}$ such that $\underline{v'' : t''}$. Then we can apply induction hypothesis on derivation $\mathcal{MS'}$, and get our target result.

$\square$

We have a lemma on the auxiliary function if-fail:

Lemma E.5. *If we have $v' = \text{if-fail}(vfres, v'')$, and:*

1. *If $vfres = \textbf{success}\ v$, then we have $v : t$ for some type t*
2. *We have $v'' : t''$ for somet type t''*

Then there exists a type t' such that $v' : t'$

Proof. Straightforwardly by case analysis on *vfres* $\square$

Similarly, we have a lemma on the auxiliary function vcombine

Lemma E.6. *If we have $vfres\star'' = \text{vcombine}(vfres, vfres\star', v, \underline{v'})$, and:*

1. *If $vfres = \textbf{success}\ v'''$, then we have $v''' : t'''$ for some type t*
2. *If $vfres\star' = \textbf{success}\ \underline{v''''}$, then we $\underline{v'''' : t''''}$ for some typing sequence $\underline{t''''}$*
3. *We have $v : t$ for somet type t*
4. *We have $\underline{v' : t'}$ for somet type t'*

Then if $vfres\star'' = \textbf{success}\ \underline{v''}$ there exists a type sequence $\underline{t''}$ such that $\underline{v'' : t''}$

Proof. Straightforwardly by case analysis on *vfres* and *vfres*$\star'$ and using Lemma E.5. $\square$

We have a lemma on the reconstruct derivation:

Lemma E.7. *If we have* $\overset{\mathcal{RC}}{\textit{\textbf{recons}}\ v\ \textit{\textbf{using}}\ \underline{v'}\ \textit{\textbf{to}}\ rcres}$ *,* $v : t$ *for some type* t*, and* $\underline{v' : t'}$ *for some type sequence* $\underline{t'}$*, then when* $rcres = \textbf{success}\ v''$ *there exists a type* t'' *such that if* $v'' : t''$

Proof. By induction on the derivation $\mathcal{RC}$:

- Cases $\mathcal{RC}$ = RC-Val-Err, $\mathcal{RC}$ = RC-Cons-Err, $\mathcal{RC}$ = RC-List-Err, $\mathcal{RC}$ = RC-Set-Err, $\mathcal{RC}$ = RC-Map-Err, $\mathcal{RC}$ = RC-Bot-Err hold trivially.
- Case $\mathcal{RC}$ = RC-Val-Sucs holds using the premises.
- Case $\mathcal{RC}$ = RC-Cons-Sucs holds using the premises and rule T-Constructor.
- Case $\mathcal{RC}$ = RC-List-Sucs holds using the premises and rule T-List.
- Case $\mathcal{RC}$ = RC-Set-Sucs holds using the premises and rule T-Set.
- Case $\mathcal{RC}$ = RC-Map-Sucs holds using the premises and rule T-Map.
- Case $\mathcal{RC}$ = RC-Bot-Sucs holds using the premises and rule T-Bot.

□

We now have a series of mutually inductive lemmas with our Theorem 6.2, since the operational semantics rules are mutually inductive themselves. The lemmas are Lemma E.8, Lemma E.9, Lemma E.10, Lemma E.11, Lemma E.12, Lemma E.13, Lemma E.14, Lemma E.15, Lemma E.16, and Lemma E.17.

Lemma E.8. *If* $\overset{\mathcal{ES}}{\underline{e}; \sigma \xRightarrow[\text{expr}\star]{} vres\star\,; \sigma'}$ *and there exists a type* t *such that* $v : t$ *for each value in the input store* $v \in \operatorname{img} \sigma$*, then*

1. *There exists a type* t' *such that* $v' : t'$ *for each value in the result store* $v' \in \operatorname{img} \sigma'$*.*
2. *If the result value* $vres\star$ *is* $\textbf{success}\ \underline{v''}$*, then there exists a type sequence* $\underline{t''}$ *such that* $\underline{v'' : t''}$*.*

3. *If the result value $vres\star$ is either* **return** v'', *or* **throw** v'', *then there exists a type t'' such that $v'' : t''$*

Proof. By induction on the derivation $\mathcal{ES}$:

- Case $\mathcal{ES} = $ ES-Emp holds directly using premises.
- Cases $\mathcal{ES} = $ ES-More, $\mathcal{ES} = $ ES-Exc1 and $\mathcal{ES} = $ ES-Exc2 hold directly from the induction hypotheses (including the one given by Theorem 6.2).

□

Lemma E.9. *If* $\overset{\mathcal{EE}}{e; \underline{\rho}; \sigma \xRightarrow[\text{each}]{} vres; \sigma'}$ *, there exists a type t such that $v : t$ for each value in the input store $v \in \operatorname{img} \sigma$, and there exists a type t such that $v : t$ for each value in an environment $v \in \operatorname{img} \rho_i$ in the environment sequence $\underline{\rho}$ then*

1. *There exists a type t' such that $v' : t'$ for each value in the result store $v' \in \operatorname{img} \sigma'$.*
2. *If the result value $vres$ is either* **success** v'', **return** v'', *or* **throw** v'', *then there exists a type t'' such that $v'' : t''$*

Proof. By induction on the derivation $\mathcal{EE}$:

- Case $\mathcal{EE} = $ EE-Emp holds directly using the premises.
- Cases $\mathcal{EE} = $ EE-More-Sucs, $\mathcal{EE} = $ EE-More-Break and $\mathcal{EE} = $ EE-More-Exc hold directly from the induction hypotheses (including Theorem 6.2), and the (trivial) facts that if any store σ is well-typed then $\sigma \setminus \mathbb{X}$ is well-typed for any set of variables $\mathbb{X}$ and $\sigma\rho$ is well-typed for any well-typed environment ρ.

□

Lemma E.10. *If* $\overset{\mathcal{G}}{g; \sigma \xRightarrow[\text{gexpr}]{} envres; \sigma'}$ *, and there exists a type t such that $v : t$ for each value in the input store $v \in \operatorname{img} \sigma$ then*

1. *There exists a type t' such that $v' : t'$ for each value in the result store $v' \in \operatorname{img} \sigma'$.*

2. *If the result value envres is* **success** $\underline{\rho}$*, then there exists a type t'' for each value in an environment $v'' \in \rho_i$ in the environment sequence $\underline{\rho}$ such that $v'' : t''$*

3. *If the result value envres is either* **return** v'''*, or* **throw** v'''*, then there exists a type t''' such that $v''' : t'''$*

Proof. By induction on the derivation $\mathcal{G}$:

- Cases $\mathcal{G} =$ G-Pat-Sucs and $\mathcal{G} =$ G-Pat-Exc hold directly by induction hypothesis given by Theorem 6.2, and then using Lemma E.3 if necessary.
- Cases $\mathcal{G} =$ G-Enum-List, $\mathcal{G} =$ G-Enum-Set and $\mathcal{G} =$ G-Enum-Map hold by the induction hypothesis given by Theorem 6.2 and then using inversion on the type derivation of the result collection value to extract the type derivations of the contained values in the result environments (from rules T-List, T-Set, and T-Map respectively).
- Cases $\mathcal{G} =$ G-Enum-Err and $\mathcal{G} =$ G-Enum-Exc hold directly by the induction hypothesis given by Theorem 6.2.

□

Lemma E.11. *If* $\underline{\rho}; e; \sigma \overset{\mathcal{C}}{\underset{\text{case}}{\Longrightarrow}} vres; \sigma'$ *, there exists a type t such that $v : t$ for each value in an environment $v \in \operatorname{img} \rho_i$ in the environment sequence $\underline{\rho}$, and there exists a type t such that $v : t$ for each value in the input store $v \in \operatorname{img} \sigma$, then*

1. *There exists a type t' such that $v' : t'$ for each value in the result store $v' \in \operatorname{img} \sigma'$.*

2. *If the result value vres is either* **success** v'', **return** v''*, or* **throw** v''*, then there exists a type t'' such that $v'' : t''$*

Proof. By induction on derivation $\mathcal{C}$:

- Case $\mathcal{C} =$ EC-Emp holds directly using the premises.
- Case $\mathcal{C} =$ EC-More-Fail and $\mathcal{C} =$ EC-More-Ord hold directly using induction hypotheses (including Theorem 6.2) and the facts of well-typedness of store extension by well-typed environments and well-typedness of variable removals from stores.

□

Lemma E.12. *If* $\overset{\mathcal{CS}}{\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{} vres; \sigma'}$ *, there exists a type t such that $v : t$, and there exists a type t' such that $v' : t'$ for each value in the input store $v' \in \operatorname{img} \sigma$, then*

1. *There exists a type t'' such that $v'' : t''$ for each value in the result store $v'' \in \operatorname{img} \sigma'$.*
2. *If the result value vres is either* **success** v''', **return** v''', *or* **throw** v''', *then there exists a type t''' such that $v''' : t'''$*

Proof. By induction on the derivation $\mathcal{CS}$:

- Case $\mathcal{CS} =$ ECS-Emp holds directly using the premises.
- Cases $\mathcal{CS} =$ ECS-More-Fail and $\mathcal{CS} =$ ECS-More-Fail hold using Lemma E.3 and the induction hypotheses (including Lemma E.11).

□

Lemma E.13. *If* $\overset{\mathcal{V}}{\underline{cs}; v; \sigma \xRightarrow[\text{visit}]{st} vres; \sigma'}$ *, there exists a type t such that $v : t$, and there exists a type t' such that $v' : t'$ for each value in the input store $v' \in \operatorname{img} \sigma$, then*

1. *There exists a type t'' such that $v'' : t''$ for each value in the result store $v'' \in \operatorname{img} \sigma'$.*
2. *If the result value vres is either* **success** v''', **return** v''', *or* **throw** v''', *then there exists a type t''' such that $v''' : t'''$*

Proof. By induction on the derivation $\mathcal{V}$:

- Cases $\mathcal{V} =$ EV-TD, $\mathcal{V} =$ EV-TDB, $\mathcal{V} =$ EV-OM-Eq, $\mathcal{V} =$ EV-OM-Neq and $\mathcal{V} =$ EV-OM-Exc hold by induction hypotheses (including Lemma E.14).
- Cases $\mathcal{V} =$ EV-BU, $\mathcal{V} =$ EV-BUB, $\mathcal{V} =$ EV-IM-Eq, $\mathcal{V} =$ EV-IM-Neq and $\mathcal{V} =$ EV-IM-Exc hold by induction hypotheses (including Lemma E.16).

□

Lemma E.14. *If* $\overset{\mathcal{VT}}{\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br} vres; \sigma'}$ *, there exists a type t such that v : t , and there exists a type t′ such that v′ : t′ for each value in the input store v′ ∈* img *σ, then*

1. *There exists a type t″ such that v″ : t″ for each value in the result store v″ ∈* img *σ′.*

2. *If the result value vres is either* **success** *v‴,* **return** *v‴, or* **throw** *v‴, then there exists a type t‴ such that v‴ : t‴*

Proof. By induction on the derivation $\mathcal{VT}$:

- All cases ($\mathcal{VT}$ = ETV-Break-Sucs, $\mathcal{VT}$ = ETV-Ord-Sucs1, $\mathcal{VT}$ = ETV-Ord-Sucs2, $\mathcal{VT}$ = ETV-Exc1 and $\mathcal{VT}$ = ETV-Exc2) hold by induction hypotheses (including Lemma E.11 and Lemma E.15), Lemma E.5, Lemma E.2 and Lemma E.7.

□

Lemma E.15. *If* $\overset{\mathcal{VTS}}{\underline{cs}; \underline{v}; \sigma \xRightarrow[\text{td}-\text{visit}\star]{br} vres\star; \sigma'}$ *, there exists a type sequence* $\underline{t}$ *such that* $\overline{\underline{v} : \underline{t}}$ *, and there exists a type t′ such that v′ : t′ for each value in the input store v′ ∈* img *σ, then*

1. *There exists a type t″ such that v″ : t″ for each value in the result store v″ ∈* img *σ′.*

2. *If the result value vres⋆ is* **success** $\underline{v'''}$ *then there exists a type sequence* $\underline{t'''}$ *such that* $\underline{v''' : t'''}$

3. *If the result value vres⋆ is either* **return** *v‴, or* **throw** *v‴, then there exists a type t‴ such that v‴ : t‴*

Proof. By induction on the derivation $\mathcal{VTS}$

- Case $\mathcal{VTS}$ = ETVS-Emp holds using the premises.

- Cases $\mathcal{VTS}$ = ETVS-Break, $\mathcal{VTS}$ = ETVS-More, $\mathcal{VTS}$ = ETVS-Exc1 and $\mathcal{VTS}$ = ETVS-Exc2 holds using the induction hypotheses (including Lemma E.14) and Lemma E.6.

□

Lemma E.16. *If* $\overset{\mathcal{VB}}{\underline{cs}; v; \sigma \xRightarrow[\text{bu-visit}]{br} vres; \sigma'}$ *, there exists a type t such that $v : t$, and there exists a type t' such that $v' : t'$ for each value in the input store $v' \in \operatorname{img} \sigma$, then*

1. *There exists a type t'' such that $v'' : t''$ for each value in the result store $v'' \in \operatorname{img} \sigma'$.*

2. *If the result value vres is either* **success** v''', **return** v''', *or* **throw** v''', *then there exists a type t''' such that $v''' : t'''$*

Proof. By induction on the derivation $\mathcal{VB}$:

- All cases ($\mathcal{VB}$ = EBU-No-Break-Sucs, $\mathcal{VB}$ = EBU-Break-Sucs, $\mathcal{VB}$ = EBU-Fail-Sucs, $\mathcal{VB}$ = EBU-No-Break-Exc, $\mathcal{VB}$ = EBU-Exc, and $\mathcal{VB}$ = EBU-No-BreakErr) hold by induction hypotheses (including Lemma E.11 and Lemma E.17), Lemma E.5, Lemma E.2 and Lemma E.7.

□

Lemma E.17. *If* $\overset{\mathcal{VBS}}{\underline{cs}; \underline{v}; \sigma \xRightarrow[\text{bu-visit}\star]{br} vres\star; \sigma'}$ *, there exists a type sequence $\underline{t}$ such that $\underline{v : t}$, and there exists a type t' such that $v' : t'$ for each value in the input store $v' \in \operatorname{img} \sigma$, then*

1. *There exists a type t'' such that $v'' : t''$ for each value in the result store $v'' \in \operatorname{img} \sigma'$.*

2. *If the result value vres$\star$ is* **success** $\underline{v'''}$ *then there exists a type sequence $\underline{t'''}$ such that $\underline{v''' : t'''}$*

3. *If the result value vres$\star$ is either* **return** v''', *or* **throw** v''', *then there exists a type t''' such that $v''' : t'''$*

Proof. By induction on the derivation $\mathcal{VBS}$

- Case $\mathcal{VBS}$ = EBUS-Emp holds using the premises.

- Cases $\mathcal{VBS}$ = EBUS-Break, $\mathcal{VBS}$ = EBUS-More, $\mathcal{VBS}$ = EBUS-Exc1 and $\mathcal{VBS}$ = EBUS-Exc2 holds using the induction hypotheses (including Lemma E.16) and Lemma E.6.

$\square$

Theorem 6.2 (Strong typing)**.** *Assume that semantic unary* $[\![\ominus]\!]$ *and binary operators* $[\![\oplus]\!]$ *are strongly typed. If* $\overset{\mathcal{E}}{e;\sigma \underset{\text{expr}}{\Longrightarrow} vres;\sigma'}$ *and there exists a type* t *such that* $\overset{\mathcal{T}}{v : t}$ *for each value in the input store* $v \in \operatorname{img} \sigma$*, then*

1. *There exists a type* t' *such that* $v' : t'$ *for each value in the result store* $v' \in \operatorname{img} \sigma'$.
2. *If the result value* $vres$ *is either* **success** v'', **return** v'', *or* **throw** v'', *then there exists a type* t'' *such that* $v'' : t''$.

Proof. By induction on the derivation $\mathcal{E}$:

- Cases $\mathcal{E}$ = E-Val, $\mathcal{E}$ = E-Var-Sucs, $\mathcal{E}$ = E-Var-Err, $\mathcal{E}$ = E-Break, $\mathcal{E}$ = E-Continue and $\mathcal{E}$ = E-Fail hold directly using the premises.
- Cases $\mathcal{E}$ = E-Un-Exc1, $\mathcal{E}$ = E-Bin-Exc1, $\mathcal{E}$ = E-Bin-Exc2, $\mathcal{E}$ = E-Cons-Err, $\mathcal{E}$ = E-Cons-Exc, $\mathcal{E}$ = E-List-Err, $\mathcal{E}$ = E-List-Exc, $\mathcal{E}$ = E-Set-Err, $\mathcal{E}$ = E-Set-Exc, $\mathcal{E}$ = E-Map-Err, $\mathcal{E}$ = E-Map-Exc, $\mathcal{E}$ = E-Lookup-Err, $\mathcal{E}$ = E-Lookup-Exc1, $\mathcal{E}$ = E-Lookup-Exc2, $\mathcal{E}$ = E-Update-Err1, $\mathcal{E}$ = E-Update-Err2, $\mathcal{E}$ = E-Update-Exc1, $\mathcal{E}$ = E-Update-Exc2, $\mathcal{E}$ = E-Update-Exc3, $\mathcal{E}$ = E-Call-Arg-Err, $\mathcal{E}$ = E-Call-Arg-Exc, $\mathcal{E}$ = E-Ret-Sucs, $\mathcal{E}$ = E-Ret-Exc, $\mathcal{E}$ = E-Asgn-Err, $\mathcal{E}$ = E-Asgn-Exc, $\mathcal{E}$ = E-If-True, $\mathcal{E}$ = E-If-False, $\mathcal{E}$ = E-If-Err, $\mathcal{E}$ = E-If-Exc, $\mathcal{E}$ = E-Switch-Sucs, $\mathcal{E}$ = E-Switch-Exc1, $\mathcal{E}$ = E-Switch-Exc2, $\mathcal{E}$ = E-Visit-Sucs, $\mathcal{E}$ = E-Visit-Fail, $\mathcal{E}$ = E-Visit-Exc1, and $\mathcal{E}$ = E-Visit-Exc2, $\mathcal{E}$ = E-For-Sucs, $\mathcal{E}$ = E-For-Exc, $\mathcal{E}$ = E-While-True-Sucs, $\mathcal{E}$ = E-While-Exc1, $\mathcal{E}$ = E-While-Exc2, $\mathcal{E}$ = E-While-Err, $\mathcal{E}$ = E-Solve-Eq, $\mathcal{E}$ = E-Solve-Neq, $\mathcal{E}$ = E-Solve-Exc, $\mathcal{E}$ = E-Solve-Err, , $\mathcal{E}$ = E-Thr-Sucs, $\mathcal{E}$ = E-Thr-Err, $\mathcal{E}$ = E-Fin-Sucs, $\mathcal{E}$ = E-Fin-Exc, and $\mathcal{E}$ = E-Try-Ord hold by induction hypotheses (including Lemma E.8, Lemma E.10, Lemma E.12 and Lemma E.13).
- Case $\mathcal{E}$ = E-Un-Sucs holds by its induction hypothesis and strong typing of $[\![\ominus]\!]$.

- Case $\mathcal{E} =$ E-Bin-Sucs holds by its induction hypothesis and strong typing of $[\![\oplus]\!]$.
- Case $\mathcal{E} =$ E-Cons-Sucs holds by its induction hypothesis given by Lemma E.8, and then by using T-Constructor with the provided typing premises.
- Case $\mathcal{E} =$ E-List-Sucs holds by its induction hypothesis given by Lemma E.8, and then by using T-List.
- Case $\mathcal{E} =$ E-Set-Sucs holds by its induction hypothesis given by Lemma E.8, and then by using T-Set.
- Case $\mathcal{E} =$ E-Map-Sucs holds by its induction hypothesis given by Lemma E.8, and then by using T-Map.
- Case $\mathcal{E} =$ E-Lookup-Sucs holds by its induction hypotheses and inversion of $\mathcal{T}$ to T-Map.
- Case $\mathcal{E} =$ E-Lookup-NoKey holds by its induction hypotheses and using T-Cons on the definition of NoKey.
- Case $\mathcal{E} =$ E-Update-Sucs holds by its induction hypotheses and by using T-Map to reconstruct the type derivation for the result.
- Cases $\mathcal{E} =$ E-Call-Sucs, $\mathcal{E} =$ E-Call-Res-Exc, $\mathcal{E} =$ E-Call-Res-Err1, and $\mathcal{E} =$ E-Call-Res-Err2 hold by induction hypotheses (including Lemma E.8) and by using the fact that extracting variables from and extending well-typed stores produces well-typed stores.
- Cases $\mathcal{E} =$ E-Asgn-Sucs and $\mathcal{E} =$ E-Try-Catch hold by induction hypotheses and the fact that extending well-typed stores with well-typed environments preserves well-typedness.
- Cases $\mathcal{E} =$ E-Switch-Fail, $\mathcal{E} =$ E-While-False, and $\mathcal{E} =$ E-While-True-Break hold by induction hypotheses and using T-Void.
- Cases $\mathcal{E} =$ E-Block-Sucs, $\mathcal{E} =$ E-Block-Exc hold by the induction hypothesis given by Lemma E.8 and the fact that removing variables from a well-typed store produces a well-typed store.

$\square$

We will for Theorem 6.3, also need to state some lemmas. We will for the proof focus mainly on cases where the induction hypothesis does not **timeout**, since if it does it is trivially possible to construct a **timeout** derivation for the result syntax.

First, we need a lemma that specifies that the sequence of values produced by the children is strictly smaller than the input value. Let $v \prec v'$ denote the relation that v is syntactically contained in v', then our target property is specified in Lemma E.18.

Lemma E.18. *If* $\underline{v'} = \text{children}(v)$ *then* $v'_i \prec v$ *for all* i*.*

Proof. Directly by induction on v. □

We have a lemma for progress on reconstruction:

Lemma E.19. *It is possible to construct a derivation* ***recons*** v ***using*** $\underline{v'}$ ***to*** rcres *for any well-typed value* v *and well-typed value sequence* $\underline{v'}$*.*

Proof. Straightforwardly by case analysis on v. □

We have two mutually recursive lemmas for pattern matching: Lemma E.20 and Lemma E.21.

Lemma E.20. *It is possible to construct a derivation* $\sigma \vdash p \overset{?}{:=} v \xRightarrow[match]{} \underline{\rho}$ *for any pattern* p*, well-typed value* v*, well-typed store* σ*.*

Proof. By induction on syntax of p:

- Case $p = vb$: We proceed by testing whether v is equal to vb:
  - Case $v = vb$ then use P-VAL-SUCS.
  - Case $v \neq vb$ then use P-VAL-FAIL.
- Case $p = x$: We proceed by testing whether x is in dom σ
  - Case $x \in$ dom σ: we proceed to test whether v is equal to $\sigma(x)$
    - * Case $v = \sigma(x)$ then use P-VAR-UNI.
    - * Case $v \neq \sigma(x)$ then use P-VAR-FAIL.
  - Case $x \notin$ dom σ then use P-VAR-BIND.

- Case $p = k(\underline{p'})$: We proceed to test whether v is equal to $k(\underline{v'})$ for some $\underline{v'}$
  - Case $v = k(v')$ then use P-Cons-Sucs using the induction hypotheses of $\underline{p'}$ with $\underline{v'}$.
  - Case $v \neq k(v')$ then use P-Cons-Fail.
- Case $p = t\ x : p'$: From our premise we know that v is well-typed, i.e. that there exists a t' such that $v : t'$. We proceed to test whether $t' <: t$.
  - Case $t' <: t$ then use P-Type-Sucs using the induction hypothesis on p' with v.
  - Case $t' \not<: t$ then use P-Type-Fail.
- Case $p = [\underline{\star p'}]$:

  We proceed to test whether $v = [\underline{v'}]$ for some value sequence $\underline{v'}$.
  - Case $v = [\underline{v'}]$ then use P-List-Sucs induction hypothesis given by Lemma E.21 on $\underline{\star p'}$ with $\underline{v'}$
  - Case $v \neq [\underline{v'}]$ then use P-List-Fail
- Case $p = \{\underline{\star p'}\}$:

  We proceed to test whether $v = \{\underline{v'}\}$ for some value sequence $\underline{v'}$.
  - Case $v = \{\underline{v'}\}$ then use P-Set-Sucs induction hypothesis given by Lemma E.21 on $\underline{\star p'}$ with $\underline{v'}$
  - Case $v \neq \{\underline{v'}\}$ then use P-Set-Fail
- Case $p = /p'$:
  - Using the induction hypothesis on p' with v, we get $\sigma \vdash p' \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho}$.
  - Now, let $\underline{v'} = \text{children}(v)$. In order to handle the self-recursive calls using $/p'$, we proceed by inner well-founded induction on the relation $\prec$ using value sequence $\underline{v'}$:
    * Using the inner induction hypothesis we get derivations $\sigma \vdash /p' \overset{?}{:=} v'_i \xRightarrow[\text{match}]{} \underline{\rho_i}$ for all i

- Finally, we use P-Deep on the derivations we got from the outer and inner induction hypotheses.

□

Lemma E.21. *It is possible to construct a derivation* $\sigma \vdash \underline{\star p} \overset{?}{:=} v \mid \mathbb{V} \xRightarrow[\mathit{match}\star]{()\quad\otimes} \underline{\rho}$ *for any pattern* p*, well-typed value* v*, well-typed store* σ*, well-typed visited value set* $\mathbb{V}$*, type-preserving partition operator* $\otimes$*.*

Proof. By induction on the syntax of $\underline{\star p}$:

- Case $\underline{\star p} = \varepsilon$

  By case analysis on $\underline{v}$:

  - Case $\underline{v} = \varepsilon$ then use PL-Emp-Both
  - Case $\underline{v} = v', \underline{v''}$ then use PL-Emp-Pat

- Case $\underline{\star p} = p', \underline{\star p''}$:

  By case analysis on $\underline{v}$:

  - Case $\underline{v} = \varepsilon$ then use PL-Emp-Val
  - Case $\underline{v} = v', \underline{v''}$ then use PL-More-Pat using induction hypotheses (including Lemma E.20).

- Case $\underline{\star p} = \star x, \underline{\star p''}$: We proceed to test whether x is in dom σ

  - Case $x \in \text{dom}\ \sigma$:
  - Case $x \notin \text{dom}\ \sigma$:

    Because of backtracking we need to do an inner induction to handle the cases which recurse to the same star patter sequence. Let $\mathbb{V}_{\text{all}} = \left\{\underline{v'} \middle| \exists \underline{v''}.\underline{v} = \underline{v'} \otimes \underline{v''}\right\}$, then we proceed by inner induction on $|\mathbb{V}_{\text{all}} - \mathbb{V}|$

    * Case $|\mathbb{V}_{\text{all}} - \mathbb{V}| = 0$:

      Then we have $\mathbb{V} = \mathbb{V}_{\text{all}}$ and so the only applicable rule is PL-More-Star-Exh since $\mathbb{V}_{\text{all}}$ covers all partitions and thus $\nexists \underline{v'}, \underline{v''}.\underline{v} = \underline{v'} \otimes \underline{v''} \wedge \underline{v'} \notin \mathbb{V}$ holds.

  * Case $|\mathbb{V}_{\text{all}} - \mathbb{V}| > 0$:
    Then there exists $\mathbb{V}' \neq \emptyset$ such that $\mathbb{V}_{\text{all}} = \mathbb{V}' \uplus \mathbb{V}$ and we can pick $\underline{v'} \in \mathbb{V}'$ such that $\underline{v} = \underline{v'} \uplus \underline{v''}$ for some $\underline{v''}$.
    We can now use PL-MORE-STAR-RE by using the outer hypothesis on $\underline{\star p}$ with $\underline{v''}$ and $\emptyset$, and the inner hypothesis on $\mathbb{V} \cup \underline{v'}$ (since the size decreases by 1)

□

We now have a series of lemmas that are mutually recursive with Theorem 6.3: Lemma E.22, Lemma E.23, Lemma E.24, Lemma E.25, Lemma E.26, Lemma E.27, Lemma E.28, Lemma E.29, Lemma E.30, and Lemma E.31.

Lemma E.22. *It is possible to construct a derivation $\underline{e}; \sigma \xRightarrow[\text{expr}\star]{}^{n} vres\star; \sigma'$, for any expression sequence $\underline{e}$, well-typed store σ and fuel n.*

Proof. By induction on n:

- Case $n = 0$ then use corresponding **timeout**-rule.
- Case $n > 0$, then by case analysis on the expression sequence $\underline{e}$:
  - Case $\underline{e} = \varepsilon$ then use ES-EMP.
  - Case $\underline{e} = e', \underline{e''}$:
    * Using induction hypothesis given by Theorem 6.3 on $n - 1$ with e' and σ we get a derivation $e'; \sigma \xRightarrow[\text{expr}]{}^{n-1} vres'; \sigma''$
      By case analysis on $vres'$:
      · Case $vres' = \textbf{success}\ v'$:
        By Theorem 6.2 we get that σ'' is well-typed.
        By induction hypothesis on $n - 1$ with $\underline{e''}$ and σ'' we get a derivation $\underline{e''}; \sigma'' \xRightarrow[\text{expr}\star]{}^{n-1} vres\star''; \sigma'$
        By case analysis on $vres\star''$:
        - Case $vres\star'' = \textbf{success}\ \underline{v''}$ then use ES-MORE
        - Case $vres\star'' = exres$ then use ES-Exc2
      · Case $vres' = exres$ then use ES-Exc1

□

Lemma E.23. *It is possible to construct a derivation* $e; \underline{\rho}; \sigma \underset{\text{each}}{\Longrightarrow}^n vres; \sigma'$ *for any expression* $\underline{e}$*, well-typed environment sequence* $\underline{\rho}$*, well-typed store* σ *and fuel* n*.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on the environment sequence $\underline{\rho}$:

  – Case $\underline{\rho} = \varepsilon$ then use EE-Emp

  – Case $\underline{\rho} = \rho', \underline{\rho''}$:
    By induction hypothesis given by Theorem 6.3 on $n-1$ with e and $\sigma\rho$ we get a derivation $e; \sigma\rho \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma''$.

    By case analysis on $vres'$:

    * Cases $vres' =$ **success** v' and $vres' =$ **continue** then use EE-More-Sucs with above derivation and the induction hypothesis on $n-1$ with $\underline{\rho''}$, e, and $\sigma''' = \sigma'' \setminus \text{dom}\, \rho'$.
    * Cases $vres' =$ **break** then use EE-More-Break
    * Cases $vres' =$ **throw** v', $vres' =$ **return** v', $vres' =$ **fail**, and $vres' =$ **error** then use EE-More-Exc

□

Lemma E.24. *It is possible to construct a derivation* $g; \sigma \underset{\text{gexpr}}{\Longrightarrow}^n envres; \sigma'$ *for any generator expression* g*, well-typed store* σ *and fuel* n*.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on g:

  – Case $g = p := e$:
    By induction hypothesis given by Theorem 6.3 on $n-1$ with e and σ we get a derivation $e; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$

    By case analysis on $vres'$:

    * Case $vres' = \mathbf{success}\ v'$ then use G-PAT-SUCS with above derivation and the derivation from applying Lemma E.20 on p with σ' and v'.
    * Case $vres' = exres$ then use G-PAT-EXC:
  - Case $g = x \leftarrow e$:
    By induction hypothesis given by Theorem 6.3 on $n-1$ with e and σ we get a derivation $e; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$

    By case analysis on $vres'$
    * Case $vres' = \mathbf{success}\ v'$
      By case analysis on v':
      · Case $v' = [\underline{v''}]$ then use $G - Enum - List$
      · Case $v' = \{\underline{v''}\}$ then use $G - Enum - Set$
      · Case $v' = (\underline{v'' : v'''})$ then use $G - Enum - Map$
      · Cases $v' = vb,\ v' = k(\underline{v''})$ and $v' = \blacksquare$ then use G-ENUM-ERR
    * Case $vres' = exres$ then use G-ENUM-EXC

□

Lemma E.25. *It is possible to construct a derivation $\underline{\rho}; e; \sigma \underset{\text{case}}{\Longrightarrow}^{n} vres; \sigma'$ for any well-typed environment sequence $\underline{\rho}$, expression e, well-typed store σ and fuel n.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:
  By case analysis on the environment sequence $\underline{\rho}$
  - Case $\underline{\rho} = 0$ then use EC-EMP.
  - Case $\underline{\rho} = \rho', \underline{\rho''}$:
    Using the induction hypothesis given by Theorem 6.3 on $n-1$ with e and σ we get a derivation $e; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$.

    By case analysis on $vres'$
    * Case $vres' = \mathbf{fail}$ then use EC-MORE-FAIL on above derivation and the derivation from applying the induction hypothesis on $n-1$ with $\underline{\rho''}$, e and σ.

* Case $vres' \neq \textbf{fail}$ then use EC-More-Ord.

□

Lemma E.26. *It is possible to construct a derivation $\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{}^{n} vres; \sigma'$ for any case sequence $\underline{cs}$, well-typed value v, well-typed store σ and fuel n.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on the case sequence $\underline{cs}$

  – Case $\underline{cs} = \varepsilon$ then use ECS-Emp

  – Case $\underline{cs} = \textbf{case}\ p \Rightarrow e, \underline{cs}'$:
  Using Lemma E.20 on p with v and σ gives us a derivation $\sigma \vdash p \overset{?}{:=} v \xRightarrow[\text{match}]{} \underline{\rho}$. By Lemma E.3 we know that $\underline{\rho}$ is well-typed.
  Using induction hypothesis given by Lemma E.25 on $n-1$ with $\underline{\rho}$, e and σ we get a derivation $\underline{\rho}; e; \sigma \xRightarrow[\text{case}]{}^{n-1} vres'; \sigma'$.
  By case analysis on $vres'$:

  * Case $vres' = \textbf{fail}$ then use ECS-More-Fail using above derivation and the derivation given by the induction hypothesis on $n-1$ with $\underline{cs}'$, v and σ.
  * Case $vres' \neq \textbf{fail}$ then use ECS-More-Ord

□

Lemma E.27. *It is possible to construct a derivation $\underline{cs}; v; \sigma \overset{st}{\underset{\text{visit}}{\Longrightarrow}}{}^{n} vres; \sigma'$ for any case sequence $\underline{cs}$, well-typed value v, well-typed store σ, traversal strategy st and fuel n.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on syntax of st

- Case $st =$ **top-down** then use EV-TD with the derivation from the induction hypothesis given by Lemma E.28 on $n-1$ with $\underline{cs}$, v, σ and **no-break**.
- Case $st =$ **top-down-break** then use EV-TDB with the derivation from the induction hypothesis given by Lemma E.28 on $n-1$ with $\underline{cs}$, v, σ and **break**.
- Case $st =$ **bottom-up** then use EV-BU with the derivation from the induction hypothesis given by Lemma E.30 on $n-1$ with $\underline{cs}$, v, σ and **no-break**.
- Case $st =$ **bottom-up-break** then use EV-BUB with the derivation from the induction hypothesis given by Lemma E.30 on $n-1$ with $\underline{cs}$, v, σ and **break**.
- Case $st =$ **outermost**:

  Using the induction hypothesis by Lemma E.28 on $n-1$ with $\underline{cs}$, v, σ and **no-break** we get a derivation $\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{\textbf{no-break}}{}^{n-1}$ $vres'; \sigma''$. We know the well-typedness of the output components from Lemma E.14.

  By case analysis on $vres'$:

  - Case $vres' =$ **success** v':

    We proceed by checking whether $v = v'$:

    - Case $v = v'$ then use EV-OM-EQ using above derivation.
    - Case $v \neq v'$ then use EV-OM-NEQ using above derivation and the derivation from the induction hypothesis given by Lemma E.28 on $n-1$ with $\underline{cs}$, v', σ'' and **no-break**.

  - Case $vres' = exres$ then use EV-OM-EXC using above derivation.

- Case $st =$ **innermost**: Using the induction hypothesis by Lemma E.30 on $n-1$ with $\underline{cs}$, v, σ and **no-break** we get a derivation $\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{\textbf{no-break}}{}^{\overline{n-1}}$ $vres'; \sigma''$. We know the well-typedness of the output components from Lemma E.16.

  By case analysis on $vres'$:

  - Case $vres' =$ **success** v':

    We proceed by checking whether $v = v'$:

· Case $v = v'$ then use EV-IM-EQ using above derivation.

· Case $v \neq v'$ then use EV-IM-NEQ using above derivation and the derivation from the induction hypothesis given by Lemma E.30 on $n-1$ with $\underline{cs}$, v', σ'' and **no-break**.

* Case $vres' = exres$ then use EV-IM-EXC using above derivation.

□

Lemma E.28. *It is possible to construct a derivation* $\underline{cs}; v; \sigma \xRightarrow[\text{td}-\text{visit}]{br}{}^{n} vres; \sigma'$ *for any* $\underline{cs}$*, well-typed value* v*, well-typed store* σ*, breaking strategy* br *and fuel* n*.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  Applying the induction hypothesis given by Lemma E.26 on $n-1$ with cs, v and $sigma$ we get a derivation $\overset{\mathcal{CS}}{\underline{cs}; v; \sigma \xRightarrow[\text{cases}]{}{}^{n-1} vres'; \sigma''}$

  We proceed to check whether $vres'$ has syntax $vfres'$ for some $vfres'$:

  – Case $vres' = vfres'$:

  We proceed to check whether $vfres' = \textbf{success}\ v'$ and $br = \textbf{break}$

  * Case $vfres' = \textbf{success}\ v'$ and $br = \textbf{break}$ then use ETV-BREAK-SUCS with $\mathcal{CS}$.

  * Case $vfres' \neq \textbf{success}\ v'$ or $br \neq \textbf{break}$:

  By Boolean logic, we have that $br \neq \textbf{break} \Rightarrow vfres' = \textbf{fail}$ is satisfied.

  Let $v'' = \text{if-fail}(vfres, v)$ and $\underline{v'''} = \text{children}(v'')$ (all well-typed from Lemma E.5 and Lemma E.2).

  By induction hypothesis given by Lemma E.15 on $n-1$ with $\underline{cs}$, $\underline{v'''}$ and σ'', we get a derivation $\overset{\mathcal{CS}'}{\underline{cs}; \underline{v'''}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br}{}^{n-1} vres''; \sigma'}$.

We know that the results are well typed from Lemma E.15.
By case analysis on $vres''$:

- Case $vfres'' = \textbf{success}\ \underline{v''''}$ then use ETV-Ord-Sucs2 with $\mathcal{CS}$, $\mathcal{CS}'$ and the result derivation from applying Lemma E.19.
- Case $vfres'' = \textbf{fail}$ then use ETV-Ord-Sucs1 with $\mathcal{CS}$, $\mathcal{CS}'$.
- Case $vfres'' = exres$ where $exres \neq \textbf{fail}$ then use ETV-Exc2 with $\mathcal{CS}$, $\mathcal{CS}'$.

– Case $vres' \neq vfres'$: Necessarily, we then have that $vres' = exres$ where $exres \neq \textbf{fail}$ and so we use ETV-Exc1 with $\mathcal{CS}$.

□

Lemma E.29. *It is possible to construct a derivation $\underline{cs}; \underline{v}; \sigma \xRightarrow[\text{td-visit}\star]{br}{}^{n}\ vres\star\ ; \sigma'$ for any $\underline{cs}$, well-typed value sequence $\underline{v}$, well-typed store σ, breaking strategy br and fuel n.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$: By case analysis on $\underline{v}$:
  - Case $\underline{v} = \varepsilon$ then use ETVS-Emp.
  - Case $\underline{v} = v', \underline{v''}$:
    By induction hypothesis given by Lemma E.28 on $n-1$ with $\underline{cs}$, v and σ we get a derivation $\overset{\mathcal{VT}}{\underline{cs}; v; \sigma \xRightarrow[\text{td-visit}]{br}{}^{n-1}\ vres''; \sigma''}$.
    We proceed by checking whether $vres''$ is syntactically a $vfres''$ for some $vfres''$:
    * Case $vres'' = vfres''$:
      We proceed by checking whether $vfres'' = \textbf{success}\ v'''$ and $br = \textbf{break}$:
      - Case $vfres'' = \textbf{success}\ v'''$ and $br = \textbf{break}$ then use ETVS-Break with $\mathcal{VT}$.

· Case $vfres'' \neq \textbf{success}\ v'''$ or $br \neq \textbf{break}$:
By Boolean logic, we have that $br \neq \textbf{break} \Rightarrow vfres'' = \textbf{fail}$ is satisfied.
By induction hypothesis on $n-1$ with $\underline{cs}$, $\underline{v''}$ and σ'' we get a derivation

$$\overset{\mathcal{VTS}}{\underline{cs}; \underline{v''}; \sigma'' \xRightarrow[\text{td}-\text{visit}\star]{br}{}^{n-1}\ vres\star'''; \sigma'}\ .$$

By case analysis on $vres\star'''$:

- Case $vres\star''' = vfres\star'''$ then use ETVS-More with $\mathcal{VT}$ and $\mathcal{VTS}$.
- Case $vres\star''' = exres$ where $exres \neq \textbf{fail}$ then use ETVS-Exc2 with $\mathcal{VT}$ and $\mathcal{VTS}$.

* Case $vres'' \neq vfres''$:
Here we then have that $vres'' = exres$, where $exres \neq \textbf{fail}$ and so we use ETVS-Exc1.

□

Lemma E.30. *It is possible to construct a derivation* $\underline{cs}; v; \sigma \xRightarrow[\text{bu}-\text{visit}]{br}{}^{n}\ vres; \sigma'$ *for any* $\underline{cs}$*, well-typed value* v*, well-typed store* σ*, breaking strategy* br *and fuel* n*.*

Proof. By case analysis on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  Let $\underline{v''} = \text{children}(v)$. We know that $\underline{v''}$ is well-typed from Lemma E.2.

  By induction hypothesis given by Lemma E.31 on $n-1$ with $\underline{cs}$, $\underline{v''}$ and σ we get a derivation $\overset{\mathcal{VBS}}{\underline{cs}; \underline{v''}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br}{}^{n-1}\ vres''; \sigma''}$. Recall that the output is well-typed from Lemma E.17.

  By case analysis on $vres\star''$:

  – Case $vres\star'' = vfres''$:
  We proceed by case analysis on $vfres''$

* Case $vfres'' = \textbf{success}\ \underline{v'''}$: We proceed by case analysis on br:
  - Case $br = \textbf{break}$ then use EBU-Break-Sucs with $\mathcal{VBS}$ and the derivation from applying Lemma E.19 on v and $\underline{v'''}$.
  - Case $br = \textbf{no-break}$:

    By applying Lemma E.19 on v and $\underline{v'''}$ we get a derivation $\overset{\mathcal{RC}}{\textbf{recons}\ v\ \textbf{using}\ \underline{v'''}\ \textbf{to}\ rcres}$.

    By case analysis on $rcres$:
    - Case $rcres = \textbf{success}\ v'$:

      By induction hypothesis given by Lemma E.26 on $n-1$ with $\underline{cs}$, v' and σ'' we get a derivation $\overset{\mathcal{CS}}{\underline{cs}; v'; \sigma'' \xRightarrow[\text{cases}]{}{}^{n-1} vres'; \sigma'}$

      By case analysis on $vres'$:
      - Case $vres' = vfres'$ then use EBU-No-Break-Sucs with $\mathcal{VBS}$, $\mathcal{RC}$ and $\mathcal{CS}$.
      - Case $exres$ then use EBU-No-Break-Exc with $\mathcal{VBS}$, $\mathcal{RC}$ and $\mathcal{CS}$.
    - Case $rcres = \textbf{error}$ then use EBU-No-Break-Err.
* Case $vfres'' = \textbf{fail}$ then use EBU-Fail-Sucs with the induction hypothesis given by Lemma E.26 on $n-1$ with $\underline{cs}$, v and σ''.

– Case $vres\star'' = exres$ where $exres \neq \textbf{fail}$ then use EBU-Exc with $\mathcal{VBS}$.

□

Lemma E.31. *It is possible to construct a derivation* $\underline{cs}; \underline{v}; \sigma \xRightarrow[\text{bu}-\text{visit}\star]{br}{}^{n} vres\star ; \sigma'$ *for any* $\underline{cs}$*, well-typed value sequence* $\underline{v}$*, well-typed store* σ*, breaking strategy* br *and fuel* n*.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on $\underline{v}$:

- Case $\underline{v} = \varepsilon$ then use EBUS-Emp.
- Case $\underline{v} = v', \underline{v''}$: By induction hypothesis given by Lemma E.30 on $n-1$ with $\underline{cs}$, v' and σ we get a derivation $\overset{\mathcal{VB}}{\underline{cs}; v'; \sigma \xRightarrow[\text{bu-visit}]{br}^{n-1} vres''; \sigma''}$.

  By case analysis on $vres''$:

  * Case $vres'' = vfres''$:

    By case analysis on $vfres''$ and br:

    · Case $vfres'' = \textbf{success}\ v'''$ and $br = \textbf{break}$: then use EBUS-Break with $\mathcal{VB}$.

    · Case $vfres'' \neq \textbf{success}\ v'''$ or $br \neq \textbf{break}$:

    By Boolean logic we have $br = \textbf{break} \Rightarrow vfres = \textbf{fail}$.

    By induction hypothesis on $n-1$ with $\underline{cs}$, $\underline{v''}$ and σ we get a derivation $\overset{\mathcal{VBS}}{\underline{cs}; \underline{v''}; \sigma'' \xRightarrow[\text{bu-visit}\star]{br}^{n-1} vres\star'; \sigma'}$.

    By case analysis on $vres\star'$:

    - Case $vres\star' = vfres\star'$ then use EBUS-More with $\mathcal{VB}$ and $\mathcal{VBS}$
    - Case $vres\star' = exres$ then use EBUS-Exc2 with $\mathcal{VB}$ and $\mathcal{VBS}$

  * Case $vres'' = exres$ then use EBUS-Exc1 with $\mathcal{VB}$.

□

Theorem 6.3 (Partial progress)**.** *It is possible to construct a derivation* $e; \sigma \xRightarrow[\text{expr}]{}^{n} vtres; \sigma'$ *for any input expression* e*, well-typed store* σ *and fuel* n*.*

Proof. By induction on n:

- Case $n = 0$ then use the corresponding **timeout**-derivation.
- Case $n > 0$:

  By case analysis on syntax e:

  - Case $e = vb$ then use T-Basic.

– Case $e = x$

We proceed by checking $x \in \text{dom}\ \sigma$:

* Case $x \in \text{dom}\ \sigma$ then use E-Var-Sucs.
* Case $x \notin \text{dom}\ \sigma$ then use E-Var-Err.

– Case $e = \ominus\ e'$:

By induction hypothesis on $n-1$ with e' and σ we get a derivation $e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$.

By case analysis on $vres'$:

* Case $vres' = \textbf{success} v$ then use E-Un-Sucs with above derivation.
* Case $vres' = exres$ then use E-Un-Exc with above derivation.

– Case $e = e_1 \oplus e_2$: By induction hypothesis on $n-1$ with e_1 and σ we get a derivation $\overset{\mathcal{E}_1}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_1; \sigma''}$.

By case analysis on $vres_1$:

* Case $vres_1 = \textbf{success}\ v_1$: By induction hypothesis on $n-1$ with e_2 and σ'' we get a derivation $\overset{\mathcal{E}_2}{e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_2; \sigma'}$.

  By case analysis on $vres_2$:

  · Case $vres_2 = \textbf{success}\ v_2$ then use E-Bin-Sucs with $\mathcal{E}_1$ and $\mathcal{E}_2$.

  · Case $vres_2 = exres$ then use E-Bin-Exc2 with $\mathcal{E}_1$ and $\mathcal{E}_2$.

* Case $vres_1 = exres$ then use E-Bin-Exc1 with $\mathcal{E}_1$.

– Case $e = k(\underline{e'})$:

Recall that all derivations in our paper are assumed to be well-scoped so there must exist a corresponding data-type at that has $k(\underline{t})$ as a constructor.

By using induction hypothesis given by Lemma E.22 on $n-1$ with $\underline{e'}$ and σ we get a derivation $\overset{\mathcal{ES}}{\underline{e'}; \sigma \underset{\text{expr}\star}{\Longrightarrow}^{n-1} vres\star'; \sigma'}$.

By case analysis on $vres\star'$:

  - Case $vres\star' = \textbf{success } \overline{v'}$:
    By Lemma E.8 we know that $\overline{v'}$ is well-typed, i.e. that we have $\overline{v' : t'}$ for some type sequence t'. We proceed to check whether all values in the sequence are non-$\blacksquare$ and each have a type t'_i that is a subtype of the target type t_i.
    - Case $\overline{v \neq \blacksquare}$ and $\overline{t' <: t}$ then use E-Cons-Sucs with $\mathcal{ES}$ and the required typing derivations.
    - Case $v_i = \blacksquare$ or $t'_i \not<: t_i$ for some i then use E-Cons-Err with $\mathcal{ES}$.
  - Case $vres\star' = exres$ then use E-Cons-Exc with $\mathcal{ES}$

- Case $e = [\underline{e'}]$:
  By induction hypothesis given by Lemma E.22 on $n - 1$ with $\underline{e'}$ and σ we get a derivation $\underline{e'}; \sigma \xRightarrow[\text{expr}\star]{}^{n-1} vres\star'; \sigma'$.
  By case analysis on $vres\star'$:
  - Case $vres\star' = \textbf{success } \overline{v}$:
    We proceed by checking whether all values $\underline{v}$ are non-$\blacksquare$:
    - Case $\overline{v \neq \blacksquare}$ then use E-List-Sucs with above derivation.
    - Case $v_i \neq \blacksquare$ for some i then use E-List-Err with above derivation.
  - Case $vres\star' = exres$ then use E-List-Exc with above derivation.

- Case $e = \{\underline{e'}\}$
  By induction hypothesis given by Lemma E.22 on $n - 1$ with $\underline{e'}$ and σ we get a derivation $\underline{e'}; \sigma \xRightarrow[\text{expr}\star]{}^{n-1} vres\star'; \sigma'$.
  By case analysis on $vres\star'$:
  - Case $vres\star' = \textbf{success } \overline{v}$:
    We proceed by checking whether all values $\underline{v}$ are non-$\blacksquare$:
    - Case $\overline{v \neq \blacksquare}$ then use E-Set-Sucs with above derivation.
    - Case $v_i \neq \blacksquare$ for some i then use E-Set-Err with above derivation.
  - Case $vres\star' = exres$ then use E-Set-Exc with above derivation.

- Case $e = (\underline{e' : e''})$

  By induction hypothesis given by Lemma E.22 on $n - 1$ with $\underline{e', e''}$ and σ we get a derivation $\underline{e', e''}; \sigma \overset{}{\underset{\text{expr}\star}{\Longrightarrow}}^{n-1} vres\star'; \sigma'$.

  By case analysis on $vres\star'$:

  * Case $vres\star' = \textbf{success}\ \underline{v, v'}$:

    We proceed by checking whether all values $\underline{v}$ and $\underline{v'}$ are non-$\blacksquare$:

    · Case $\underline{v \neq \blacksquare}$ and $\underline{v' \neq \blacksquare}$ then use E-Map-Sucs with above derivation.

    · Case $v_i \neq \blacksquare$ or $v'_i \neq \blacksquare$ for some i then use E-Map-Err with above derivation.

  * Case $vres\star' = exres$ then use E-Map-Exc with above derivation.

- Case $e = e_1[e_2]$

  By induction hypothesis on $n - 1$ with e_1 and σ we get a derivation $\overset{\mathcal{E}}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_1; \sigma''}$.

  By case analysis on $vres_1$:

  * Case $vres_1 = \textbf{success}\ v_1$:

    By case analysis on v_1:

    · Case $v_1 = (\underline{v' : v''})$:

    By induction hypothesis on $n - 1$ with e_2 and σ'' we get a derivation $\overset{\mathcal{E}'}{e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_2; \sigma'}$.

    By case analysis on $vres_2$:

    • Case $vres_2 = \textbf{success}\ v_2$:

      We proceed to check whether $\exists i. v'_i = v_2$:

      – Case $v'_i = v_2$ then use E-Lookup-Sucs with $\mathcal{E}$ and $\mathcal{E}'$.

      – Case $\nexists i. v'_i = v_2$ then use E-Lookup-NoKey with $\mathcal{E}$ and $\mathcal{E}'$.

    • Case $vres_2 = exres$ then use E-Lookup-Exc2 with $\mathcal{E}$ and $\mathcal{E}'$.

    · Case $v_1 \neq (\underline{v' : v''})$ then use E-Lookup-Err with $\mathcal{E}$.

  * Case $vres_1 = exres$ then use E-Lookup-Exc1 with $\mathcal{E}$.

- Case $e = e_1[e_2 = e_3]$

  By induction hypothesis on $n-1$ with e_1 and σ we get a derivation $e_1; \sigma \overset{\mathcal{E}}{\underset{\text{expr}}{\Longrightarrow}}{}^{n-1} \mathit{vres}_1; \sigma'''$.

  By case analysis on vres_1:

  * Case $\mathit{vres}_1 = \textbf{success}\ v_1$

    By case analysis on v_1:

    - Case $v_1 = (\underline{v' : v''})$:

      By induction hypothesis on $n-1$ with e_2 and σ''' we get a derivation $e_2; \sigma''' \overset{\mathcal{E}'}{\underset{\text{expr}}{\Longrightarrow}}{}^{n-1} \mathit{vres}_2; \sigma''$.

      By case analysis on vres_2:

      - Case $\mathit{vres}_2 = \textbf{success}\ v_2$:

        By induction hypothesis on $n-1$ with e_3 and σ'' we get a derivation $e_3; \sigma'' \overset{\mathcal{E}''}{\underset{\text{expr}}{\Longrightarrow}}{}^{n-1} \mathit{vres}_3; \sigma'$.

        By case analysis on vres_3:

        - Case $\mathit{vres}_3 = \textbf{success}\ v_3$:

          We proceed to check whether v_2 and v_3 are non-$\blacksquare$:

          * Case $v_2 \neq \blacksquare$ and $v_3 \neq \blacksquare$ then use E-Update-Sucs with $\mathcal{E}$, $\mathcal{E}'$ and $\mathcal{E}''$.
          * Case $v_2 = \blacksquare$ or $v_3 = \blacksquare$ then use E-Update-Err2 with $\mathcal{E}$, $\mathcal{E}'$ and $\mathcal{E}''$.

        - Case $\mathit{vres}_3 = \mathit{exres}$ then use E-Update-Exc3 with $\mathcal{E}$ and $\mathcal{E}'$.

      - Case $\mathit{vres}_2 = \mathit{exres}$ then use E-Update-Exc2 on $\mathcal{E}$ and $\mathcal{E}'$.

    - Case $v_1 \neq (\underline{v' : v''})$ then use E-Update-Err1 with $\mathcal{E}$.

  * Case $\mathit{vres}_1 = \mathit{exres}$ then use E-Update-Exc1 with $\mathcal{E}$.

- Case $e = f(\underline{e'})$

  By induction hypothesis given by Lemma E.22 on $n-1$ with $\underline{e'}$ and σ we get a derivation $\underline{e'}; \sigma \overset{\mathcal{ES}}{\underset{\text{expr}\star}{\Longrightarrow}}{}^{n-1} \mathit{vres}\star''; \sigma''$

  By case analysis on $\mathit{vres}\star''$:

* Case $vres\star'' = \textbf{success}\ \underline{v''}$
  Recall that we assume that our function calls are well-scoped and so there must exist a corresponding function definition $\textbf{fun}\ t'\ f(\underline{t\ x}) = e''$. By Lemma E.8 we know that we have $\underline{v'' : t''}$ for some type sequence $\underline{t''}$.
  We proceed to check whether $\underline{t'' <: t}$:
  · Case $\underline{t'' <: t}$
    Let $\underline{\textbf{global}\ t_y\ y}$ represent all global variable definitions.
    By induction hypothesis on $n-1$ with e'' and $[\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v''}]$ we get a derivation
    $$\underset{\mathcal{E}}{\underbrace{e'; [\underline{y \mapsto \sigma''(y)}, \underline{x \mapsto v''}] \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'''}} \ .$$
    By case analysis on $vres'$:
    - Case $vres' = \textbf{success}\ v'$ or $vres' = \textbf{return}\ v'$.
      By Theorem 6.2 we know that $v' : t'''$ for some type t'''. We proceed to check whether $t''' <: t'$:
      – Case $t''' <: t'$ then use E-CALL-RES-SUCS with $\mathcal{ES}$ and $\mathcal{E}$.
      – Case $t''' \not<: t'$ then use E-CALL-RES-ERR1 with $\mathcal{ES}$ and $\mathcal{E}$.
    - Case $vres' = \textbf{throw}\ v'$ then use E-CALL-RES-EXC with $\mathcal{ES}$ and $\mathcal{E}$
    - Case $vres' \in \{\textbf{break}, \textbf{continue}, \textbf{fail}, \textbf{error}\}$ then use E-CALL-RES-ERR2 with $\mathcal{ES}$ and $\mathcal{E}$.
  · Case $t''_i \not<: t_i$ for some i then use E-CALL-ARG-ERR with $\mathcal{ES}$.
* Case $vres\star'' = exres$ then use E-CALL-ARG-EXC with $\mathcal{ES}$.

– Case $e = \textbf{return}\ e'$
  By induction hypothesis on $n-1$ with e' and σ we get a derivation $e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$.
  * Case $vres' = \textbf{success}\ v$ then use E-RET-SUCS with above derivation.
  * Case $vres' = exres$ then use E-RET-EXC with above derivation.

- Case $e = (x = e')$

  Recall that our definitions are assumed to be well-scoped and so there must exists either a **local** $t\ x$ or **global** $t\ x$ declaration for the variable (with no overshadowing).

  By induction hypothesis on $n-1$ with e' and σ we get a derivation $e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma''$.

  By case analysis on $vres'$:

  * Case $vres' =$ **success** v: By Theorem 6.2 we know that there exists a t' such that $v : t'$.

    We proceed by checking whether $t' <: t$:

    · Case $t' <: t$ then use E-Asgn-Sucs with above evaluation and typing derivations.

    · Case $t' \not<: t$ then use E-Asgn-Err with above evaluation and typing derivations.

  * Case $vres' = exres$ then use E-Asgn-Exc with above derivation.

- Case $e =$ **if** e_1 **then** e_2 **else** e_3

  By induction hypothesis on $n-1$ with e_1 and σ we get a derivation $\overset{\mathcal{E}}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres''; \sigma''}$.

  By case analysis on $vres''$:

  * Case $vres'' =$ **success** v''

    By case analysis on v'':

    · Case $v'' = \text{false}()$ then use E-If-False with $\mathcal{E}$ and the derivation from the induction hypothesis on $n-1$ with e_2 and σ''.

    · Case $v'' = \text{true}()$ then use E-If-True with $\mathcal{E}$ and the derivation from the induction hypothesis on $n-1$ with e_3 and σ''.

    · Case $v'' \neq \text{true}()$ and $v'' \neq \text{false}()$ then use E-If-Err with $\mathcal{E}$.

  * Case $vres'' = exres$ then use E-If-Exc with $\mathcal{E}$.

- Case $e =$ **switch** e' **do** $\overline{cs}$

  By induction hypothesis on $n-1$ with e' and σ we get a derivation $\overset{\mathcal{E}}{e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres''; \sigma''}$.

By case analysis on *vres''*:

* Case $vres'' = \textbf{success}\ v''$
  By induction hypothesis given by Lemma E.26 on $n-1$ with $\underline{cs}$, v'' and σ'' we get a derivation $\overset{\mathcal{CS}}{\underline{cs}; v''; \sigma'' \underset{\text{cases}}{\Longrightarrow}^{n-1} vres'; \sigma'}$.
  By case analysis on *vres'*:
  · Case $vres' = \textbf{success}\ v'$ then use E-Switch-Sucs with $\mathcal{E}$ and $\mathcal{CS}$.
  · Case $vres' = \textbf{fail}$ then use E-Switch-Fail with $\mathcal{E}$ and $\mathcal{CS}$.
  · Case $vres' = exres$ where $exres \neq \textbf{fail}$ then use E-Switch-Exc2 with $\mathcal{E}$ and $\mathcal{CS}$.
* Case $vres'' = exres$ then use E-Switch-Exc1 with $\mathcal{E}$.

– Case $e = st\ \textbf{visit}\ e'\ \textbf{do}\ \underline{cs}$:
  By induction hypothesis on $n-1$ with e' and σ we get a derivation $\overset{\mathcal{E}}{e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres''; \sigma''}$.
  By case analysis on *vres''*:
  * Case $vres'' = \textbf{success}\ v''$
    By induction hypothesis given by Lemma E.27 on $n-1$ with $\underline{cs}$, v'' and σ'' we get a derivation $\overset{\mathcal{V}}{\underline{cs}; v''; \sigma'' \overset{st}{\underset{\text{visit}}{\Longrightarrow}}^{n-1} vres'; \sigma'}$.
    By case analysis on *vres'*:
    · Case $vres' = \textbf{success}\ v'$ then use E-Visit-Sucs with $\mathcal{E}$ and $\mathcal{V}$.
    · Case $vres' = \textbf{fail}$ then use E-Visit-Fail with $\mathcal{E}$ and $\mathcal{V}$.
    · Case $vres' = exres$ where $exres \neq \textbf{fail}$ then use E-Visit-Exc2 with $\mathcal{E}$ and $\mathcal{V}$.
  * Case $vres'' = exres$ then use E-Visit-Exc1 with $\mathcal{E}$.

– Case $e = \textbf{break}$ then use E-Break.

– Case $e = \textbf{continue}$ then use E-Continue.

– Case $e = \textbf{fail}$ then use E-Fail.

- Case $e = \textbf{local}\ \underline{t\ x}\ \textbf{in}\ \underline{e'}\ \textbf{end}$ then use either E-Block-Sucs (when it produces a successful result) or E-Block-Exc (otherwise) with the derivation of the induction hypothesis given by Lemma E.22 on $n-1$ with $\underline{e'}$ and σ.
- Case $e = \textbf{for}\ g\ e'$:

  By induction hypothesis given by Lemma E.24 on $n-1$ with g and σ we get a derivation $g; \sigma \underset{\text{gexpr}}{\Longrightarrow}^{n-1} \mathit{envres}; \sigma''$.

  By case analysis on *envres*:

  * Case $\mathit{envres} = \textbf{success}\ \underline{\rho}$ then use E-For-Sucs with above derivation and the derivation from the induction hypothesis given by Lemma E.23 on $n-1$ with e', $\underline{\rho}$ and σ''.
  * Case $\mathit{envres} = \mathit{exres}$ then use E-For-Exc with above derivation.
- Case $e = \textbf{while}\ e_1\ e_2$

  By induction hypothesis on $n-1$ with e_1 and σ we get a derivation: $\overset{\mathcal{E}}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} \mathit{vres}''; \sigma''}$.

  By case analysis on vres'':

  * Case $\mathit{vres}'' = \textbf{success}\ v''$

    By case analysis on v'':

    · Case $v'' = \text{false}()$ then use E-While-False with $\mathcal{E}$.

    · Case $v'' = \text{true}()$:

    By induction hypothesis on $n-1$ with e_2 and σ'' we get a derivation $\overset{\mathcal{E}'}{e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow}^{n-1} \mathit{vres}'''; \sigma'''}$.

    By case analysis on vres''':

    - Case $\mathit{vres}''' = \textbf{success}\ v'''$ or $\mathit{vres}''' = \textbf{continue}$ then use E-While-True-Sucs with $\mathcal{E}$, $\mathcal{E}'$ and the derivation from the induction hypothesis on $n-1$ with **while** $e_1\ e_2$ and σ'''.
    - Case $\mathit{vres}''' = \textbf{break}$ then use E-While-True-Break with $\mathcal{E}$ and $\mathcal{E}'$.
    - Case $\mathit{vres}''' = \mathit{exres} \in \{\textbf{throw}\ v''', \textbf{return}\ v''', \textbf{fail}, \textbf{error}\}$ then use E-While-Exc2 with $\mathcal{E}$ and $\mathcal{E}'$.

      · Case $v'' \neq \text{true}()$ and $v'' \neq \text{false}()$ then use E-While-Err

    * Case $vres'' = exres$ then use E-While-Exc1 with $\mathcal{E}$.

- Case $e = \textbf{solve}\ \underline{x}\ e'$:

  By induction hypothesis on $n-1$ with e' and σ we get a derivation $\overset{\mathcal{E}}{e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres''; \sigma''}$.

  By case analysis on $vres''$

    * Case $vres'' = \textbf{success}\ v''$:

      We proceed by checking whether the variable sequence $\underline{x}$ is both in old and new stores.

      · Case $\underline{x} \subseteq \text{dom}\ \sigma \cap \text{dom}\ \sigma''$:

        We then check whether any value in $\underline{x}$ has changed:

        • Case $\underline{\sigma(x) = \sigma''(x)}$ then use E-Solve-Eq with $\mathcal{E}$.

        • Case $\sigma(x_i) \neq \sigma''(x_i)$ for some i then use E-Solve-Neq with $\mathcal{E}$ and the derivation from the induction hypothesis on $n-1$ with $\textbf{solve}\ \underline{x}\ e'$ and σ''.

      · Case $x_i \notin \text{dom}\ \sigma \cap \text{dom}\ \sigma''$ for some i then use E-Solve-Err with $\mathcal{E}$.

    * Case $vres'' = exres$ then use E-Solve-Exc with $\mathcal{E}$.

- Case $e = \textbf{throw}\ e'$:

  By induction hypothesis on $n-1$ with e' and σ we get a derivation $e'; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres'; \sigma'$.

    * Case $vres' = \textbf{success}\ v$ then use E-Thr-Sucs with above derivation.

    * Case $vres' = exres$ then use E-Thr-Exc with above derivation.

- Case $e = \textbf{try}\ e_1\ \textbf{catch}\ x \Rightarrow e_2$:

  By induction hypothesis on $n-1$ with e_1 and σ we get a derivation $\overset{\mathcal{E}}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_1; \sigma''}$.

  By case analysis on $vres_1$:

    * Case $vres_1 = \textbf{throw}\ v_1$ then use E-TRY-CATCH with $\mathcal{E}$ and the derivation from the induction hypothesis on $n-1$ with e_2 and $\sigma''[x \mapsto v_1]$.
    * Case $vres_1 \neq \textbf{throw}\ v_1$ then use E-TRY-ORD with $\mathcal{E}$.

- Case $e = \textbf{try}\ e_1\ \textbf{finally}\ e_2$

  By induction hypothesis on $n-1$ with e_1 and σ we get a derivation $\overset{\mathcal{E}}{e_1; \sigma \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_1; \sigma''}$.

  By induction hypothesis on $n-1$ with e_2 and σ'' we get a derivation $\overset{\mathcal{E}'}{e_2; \sigma'' \underset{\text{expr}}{\Longrightarrow}^{n-1} vres_2; \sigma'}$.

  By case analysis on $vres_2$:

    * Case $vres_2 = \textbf{success}\ v_2$ then use E-FIN-SUCS with $\mathcal{E}$ and $\mathcal{E}'$
    * Case $vres_2 = exres$ then use E-FIN-EXC with $\mathcal{E}$ and $\mathcal{E}'$.

$\square$

Theorem 6.4 (Terminating expressions)**.** *There exists n such that derivation* $\overset{\mathcal{E}}{e_{\text{fin}}; \sigma \underset{\text{expr}}{\Longrightarrow}^{n} vres; \sigma'}$ *has a result vres which is not* **timeout** *for expression* e_{fin} *in the terminating subset.*

Proof. The proof proceeds similarly to Theorem 6.3, except that instead of doing the induction on n—which we know need to provide—we do the induction on the relevant syntactic element starting with the e_{fin} for this theorem. The only major complication is that for the **bottom-up** visit rules, we need to do an inner well-founded induction on the $\prec$ relation on values when traversing the children in order to terminate.

The result n is simply taking to be $n' = 1 + n$ where n is the maximal fuel used in a sub-term. $\square$

Appendix F

Rascal Light Verification Proofs

Theorem 7.1. *Assuming Galois connection for element domains* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$ *then we have a Galois connection for the set shape domain* $\wp(\wp(\mathrm{E})) \underset{\alpha_{\widehat{\mathrm{SS}}}}{\overset{\gamma_{\widehat{\mathrm{SS}}}}{\leftrightarrows}} \widehat{\mathrm{SetShape}}(\widehat{\mathrm{E}})$

Proof. Since $\widehat{\mathrm{SetShape}}(\widehat{\mathrm{E}})$ is a reduced product domain, then this property follows directly from the element domain Galois connection, and the interval domain Galois connection. □

Theorem 7.2. *Assuming a Galois connection for the leaf elements* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$ *then we have a Galois connection for the data elements* $\wp(\mathrm{Data}) \underset{\alpha_{\widehat{\mathrm{DS}}}}{\overset{\gamma_{\widehat{\mathrm{DS}}}}{\leftrightarrows}} \widehat{\mathrm{DataShape}}(\widehat{\mathrm{E}})$

Proof. Follows directly by lifting from Theorem 7.3. □

Theorem 7.3. *Given a Galois connection* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$, *then there is a Galois connection* $\mathrm{PData}(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{R}}}}{\overset{\gamma_{\widehat{\mathrm{R}}}}{\leftrightarrows}} \widehat{\mathrm{Refinement}}(\widehat{\mathrm{E}})$

Proof Sketch. In essence this Galois connection signifies a conversion between syntactic definitions and semantics of data type refinements, except that the contained values have to respectively be in domains $\widehat{\mathrm{E}}$ and E respectively, whose Galois connection follows from the premise. Note, that in case of fixed-point domains used in $\widehat{\mathrm{E}}$ there is a possible hidden recursion between the operations on $\widehat{\mathrm{E}}$ and $\widehat{\mathrm{R}}$, but these are in such case also interpreted as least fixed-point equations whose validity depends on Theorem 7.5. □

Theorem 7.4. *Given* $\forall i \left(\wp\left(\mathrm{A}_i\right) \underset{\alpha_{\widehat{\mathrm{A}}_i}}{\overset{\gamma_{\widehat{\mathrm{A}}_i}}{\leftrightarrows}} \widehat{\mathrm{A}}_i \right)$ *then* $\wp\left(\biguplus \underline{\mathrm{A}}\right) \underset{\alpha_{\oplus}}{\overset{\gamma_{\oplus}}{\leftrightarrows}} \bigoplus(\underline{\widehat{\mathrm{A}}})$

Proof. We have to show $eas \subseteq \gamma_{\oplus}(\widehat{ea})$ if and only if $\alpha_{\oplus}(eas) \sqsubseteq \widehat{ea}$. We do this by showing it holds in both directions, by case analysis on $\widehat{ea}$ and inversion of the definition $\sqsubseteq_{\oplus}$, then delegating to the relevant Galois connection. □

Theorem 7.5. *If that for continuous parametrized domains* F, $\widehat{\mathrm{F}}$ *it holds that for all* E, $\widehat{\mathrm{E}}$ *such that* $\wp\left(\mathrm{E}\right) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$ *then* $\wp\left(\mathrm{F}(\mathrm{E})\right) \underset{\alpha_{\widehat{\mathrm{F}}}}{\overset{\gamma_{\widehat{\mathrm{F}}}}{\leftrightarrows}} \widehat{\mathrm{F}}(\widehat{\mathrm{E}})$, *then* $\wp\left(\mathrm{lfp}\ \mathrm{X}.\mathrm{F}(\mathrm{X})\right) \underset{\alpha_{\widehat{\mathrm{Fix}}}}{\overset{\gamma_{\widehat{\mathrm{Fix}}}}{\leftrightarrows}} \mathrm{lfp}\ \widehat{\mathrm{X}}.\widehat{\mathrm{F}}(\widehat{\mathrm{X}})$

Proof. By definition of fixed-points[Scott, 1976; Smyth and Plotkin, 1982] we have that

$$\mathrm{lfp}\ \mathrm{X}.\mathrm{F}(\mathrm{X}) = \bigsqcup_{n \in \mathbb{N}} F^{(n)}(\mathbb{0})$$

where $\mathbb{0}$ is the empty domain. It suffices to show that for all n, there exists α' and γ' such that $\wp\left(\mathrm{F}^{(n)}(\mathbb{0})\right) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \widehat{\mathrm{F}}^{(n)}(\mathbb{0})$, which is done by induction over n:

- Case $n = 0$: We have to show $\wp\left(\mathbb{0}\right) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \mathbb{0}$ which holds vacuously.

- Case $n = k + 1$: By induction hypothesis we have

$$\wp\left(\mathrm{F}^{(k)}(\mathbb{0})\right) \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} \widehat{\mathrm{F}}^{(k)}(\mathbb{0})$$

  and we have to show

$$\left(\mathrm{F}^{(k+1)}(\mathbb{0})\right) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \widehat{\mathrm{F}}^{(k+1)}(\mathbb{0})$$

  which holds by application of the premise on the induction hypothesis.

□

Corollary 7.1. *If given a Galois connection for the parameter element domain* $\wp(\mathrm{E}) \underset{\alpha_{\widehat{\mathrm{E}}}}{\overset{\gamma_{\widehat{\mathrm{E}}}}{\leftrightarrows}} \widehat{\mathrm{E}}$*, then we have a Galois connection for the parameterized value domain* $\wp(\mathrm{PValue}(\mathrm{E})) \underset{\alpha_{\widehat{\mathrm{PVS}}}}{\overset{\gamma_{\widehat{\mathrm{PVS}}}}{\leftrightarrows}} \widehat{\mathrm{PValueShape}}(\widehat{\mathrm{E}})$

Proof. Follows directly from Proposition 7.1, Theorem 7.1, Theorem 7.2 and Theorem 7.4. □

Corollary 7.2. *We have a Galois connection* $\wp(\mathrm{Value}) \underset{\alpha_{\widehat{\mathrm{VS}}}}{\overset{\gamma_{\widehat{\mathrm{VS}}}}{\leftrightarrows}} \widehat{\mathrm{ValueShape}}$

Proof. Follows directly from Corollary 7.1 and Theorem 7.5. □

Theorem 7.6. *We have a Galois connection* $\wp(\mathrm{Store}) \underset{\alpha_{\widehat{\mathrm{Store}}}}{\overset{\gamma_{\widehat{\mathrm{Store}}}}{\leftrightarrows}} \widehat{\mathrm{Store}}$

Proof. Follows from lifting of the reduced product domain of the Boolean domain and $\widehat{\mathrm{Value}}$ abstract domain, the latter which holds from Corollary 7.2. □

Theorem 7.7. *We have a Galois connection*

$$\wp(\mathrm{ValueResult} \times \mathrm{Store}) \underset{\alpha_{\widehat{\mathrm{RS}}}}{\overset{\gamma_{\widehat{\mathrm{RS}}}}{\leftrightarrows}} \widehat{\mathrm{ResultSet}}$$

Proof Sketch. We are essentially representing an infinite set of value result-store pairs in a finite set containing abstract values partitioned by their kind of control flow. The proof can therefore be done in two steps, one mapping directly the concrete value result-store pairs to abstract value result-store pairs in the domain $\wp\left(\widehat{\mathrm{ValueResult}} \times \widehat{\mathrm{Store}}\right)$—which is holds by lifiting of Corollary 7.2 and Theorem 7.6)—and then show that this domain has a Galois connection to $\widehat{\mathrm{ResultSet}}$ which is straightforward since it simply takes the least upper bound of all subsets with a particular kind of control flow. □

Theorem 7.8 (Soundness). *For all valid expressions e, concrete stores σ and over-approximating abstract stores $\widehat{\sigma}$, so $\sigma \in \gamma_{\widehat{\mathrm{Store}}}(\widehat{\sigma})$, where we have a concrete evaluation derivation $e; \widehat{\sigma} \underset{\mathrm{expr}}{\Longrightarrow} vres; \sigma'$ and corresponding abstract evaluation derivation $e; \widehat{\sigma} \underset{a\text{-}expr}{\Longrightarrow} \widehat{Res}$ then it holds that the abstract result set properly over-approximates the concrete result value result and store, i.e., $(vres, \sigma') \in \gamma_{RS}(\widehat{Res})$.*

Proof Sketch. The soundness of our abstract interpreter concretely depends on three aspects:

1. That the abstract evaluation rules over-approximate all possible corresponding concrete evaluations. Because of the close correspondence between abstract and concrete evaluation rules in Schmidt-style abstract interpretation, this is done by showing that there is a homomorphism [Schmidt, 1998] from the concrete rules to the abstract rules, i.e. that for each concrete rule there exists a set of abstract rules that covers its evaluation.

2. That the abstract conditions and semantic operations properly over-approximate the concrete conditions and semantic operations. This is done using classical abstract domain operation analysis, as done for the theorems above.

3. That the chosen memoization strategies produce monotone results. The argument largely follows the one presented in Rosendahl [2013], but we must take into account the extension to infinite inputs. This extension is sound since we on recursion maintain monotonicity by using the least upper bound with the previous occurrence of inputs from the same partition and terminating because the partitioning is finite and the additional widening on input.

□

Appendix G

Rascal Subject programs

Negation Normal Form:

```
module NNF

data Formula = atom(str nm)
             | and(Formula l, Formula r)
             | or(Formula l, Formula r)
             | imp(Formula l, Formula r)
             | neg(Formula f);

Formula nnf(Formula phi) = top-down visit(phi) {
  case neg(or(l,r)) => and(neg(l), neg(r))
  case neg(and(l,r)) => or(neg(l), neg(r))
  case neg(imp(l,r)) => and(l, neg(r))
  case neg(neg(f)) => nnf(f)
  case imp(l,r) => or(neg(l), r)
};
```

Rename Field of Struct:

```
module RenameStructField

data Nominal = nfn() | ofn() | other();
data Package = package(map[str, Struct] structures,
                       map[str, Function] functions);
data Struct = struct(str name, map[Nominal, Field] fields);
data Field = field(Nominal name, str typ);
data Function = function(str name, str return_typ,
                         list[Parameter] parameters, Stmt body);
data Parameter = parameter(str typ, str name);
data Stmt = ifstmt(Expr cond, Stmt thenb, Stmt elseb)
          | returnstmt(Expr val)
          | assignstmt(Expr lhs, Expr rhs)
          | block(list[Stmt] stmts)
          ;
data Expr = fieldaccessexpr(Expr target, Nominal fieldname)
```

```
          | varexpr(str name)
          | functioncallexpr(Expr target, str methodname, list[Expr] args)
          ;

Package renameField(Package pkg, str st,
                    Nominal oldFieldName, Nominal newFieldName) {
    assert (st in pkg.structures) &&
           (oldFieldName in pkg.structures[st].fields) &&
           (newFieldName notin pkg.structures[st].fields) ;
    Field fieldDef = pkg.structures[st].fields[oldFieldName];
    fieldDef.name = newFieldName;
    pkg.structures[st].fields =
     delete(pkg.structures[st].fields, oldFieldName)
       + (newFieldName: fieldDef);
    return top-down visit(pkg) {
        case fieldaccessexpr(target, oldFieldName) =>
              fieldaccessexpr(target, newFieldName)
    };
}
```

Desugar Oberon

```
module DesugarOberon

data Module =
    \mod(Ident name, Declarations decls, list[Statement] body, Ident endName)
    ;

data Declarations
    = decls(list[ConstDecl] consts, list[TypeDecl] types, list[VarDecl] vars)
    ;

data ConstDecl
    = constDecl(Ident name, Expression \value)
    ;

data TypeDecl
    = typeDecl(Ident name, Type \type)
    ;

data VarDecl
    = varDecl(list[Ident] names, Type \type)
    ;

data Type
    = user(Ident name)
    ;

data MaybeExpression
    = nothing()
    | just(Expression e)
    ;

data Statement
    = assign(Ident var, Expression exp)
    | ifThen(Expression condition, list[Statement] body,
            list[ElseIf] elseIfs, list[Statement] elsePart )
    | whileDo(Expression condition, list[Statement] body)
    | skip()
    | forDo(Ident name, Expression from, Expression to,
           MaybeExpression by, list[Statement] body)
    | caseOf(Expression exp, list[Case] cases, list[Statement] elsePart)
    | begin(list[Statement] body)
    ;

data Expression
    = nat(int val)
    | \trueE()
    | \falseE()
    | lookup(Ident var)
    | neg(Expression exp)
    | pos(Expression exp)
    | not(Expression exp)
    | mul(Expression lhs, Expression rhs)
    | div(Expression lhs, Expression rhs)
    | amp(Expression lhs, Expression rhs)
    | add(Expression lhs, Expression rhs)
    | sub(Expression lhs, Expression rhs)
    | or(Expression lhs, Expression rhs)
    | eq(Expression lhs, Expression rhs)
    | neq(Expression lhs, Expression rhs)
    | lt(Expression lhs, Expression rhs)
    | gt(Expression lhs, Expression rhs)
    | leq(Expression lhs, Expression rhs)
    | geq(Expression lhs, Expression rhs)
    ;

data Ident
    = id(str name)
;


data Case
    = guard(Expression guard, list[Statement] body)
;

data ElseIf = elseif(Expression condition, list[Statement] body);

public Module desugar(Module \mod) {
    \mod.body = flattenBegin(case2ifs(for2while((\mod.body))));
    return \mod;
}

```

```
Statement cases2if(Expression e, list[Case] cs, list[Statement] es) {
    switch (cs) {
      case []: return begin(es);
      case [c, *cs2]: {
            list[ElseIf] eis = [];
            for (c2 <- cs2)
               eis = eis + [elseif(eq(e, c2.guard), c.body)];
            return ifThen(eq(e, c.guard), c.body, eis, es);
        }
    };
}

public list[Statement] case2ifs(list[Statement] stats) {
    return visit (stats) {
        case caseOf(e, cs, es) => cases2if(e, cs, es)
    }
}

public list[Statement] for2while(list[Statement] stats) {
    return visit (stats) {
        case forDo(n, f, t, just(by), b) =>
            begin([assign(n, f), whileDo(geq(lookup(n), t),
                    b + [assign(n, add(lookup(n), by))])])
        case forDo(n, f, t, nothing(), b) =>
            begin([assign(n, f), whileDo(geq(lookup(n), t),
                    b + [assign(n, add(lookup(n), nat(1)))])])
    }
}

public list[Statement] flattenBegin(list[Statement] stats) {
    return innermost visit (stats) {
        case list[Statement] ss =>
            ({
              list[Statement] res = [];
              for (s <- ss) {
                switch (s) {
                   case begin(b): res = res + b;
                   case _: res = res + [s];
                };
              };
              res;
            })
    }
}
```

Glagol-to-PHP Expression Translation:

```
module Glagol2PHP

```

```
data Unsuppported = unsupported();

data Expression
    = integer(int intValue)
    | string(str strValue)
    | boolean(bool boolValue)
    | \list(list[Expression] values)
    | arrayAccess(Expression variable, Expression
        ↪ arrayIndexKey)
    | \map(map[Expression key, Expression \value])
    | variable(str name)
    | \bracket(Expression expr)
    | product(Expression lhs, Expression rhs)
    | remainder(Expression lhs, Expression rhs)
    | division(Expression lhs, Expression rhs)
    | addition(Expression lhs, Expression rhs)
    | subtraction(Expression lhs, Expression rhs)
    | greaterThanOrEq(Expression lhs, Expression rhs)
    | lessThanOrEq(Expression lhs, Expression rhs)
    | lessThan(Expression lhs, Expression rhs)
    | greaterThan(Expression lhs, Expression rhs)
    | equals(Expression lhs, Expression rhs)
    | nonEquals(Expression lhs, Expression rhs)
    | and(Expression lhs, Expression rhs)
    | or(Expression lhs, Expression rhs)
    | negative(Expression expr)
    | positive(Expression expr)
    | ternary(Expression condition, Expression ifThen,
        ↪ Expression \else)
    | new(str artifact, list[Expression] args)
    | get(Type t)
    | invoke(str methodName, list[Expression] args)
    | invoke2(Expression prev, str methodName, list[
        ↪ Expression] args)
    | fieldAccess(str field)
    | fieldAccess2(Expression prev, str field)
    | emptyExpr()
    | this()
    ;

data Type
    = integer2()
    | string2()
    | voidValue()
    | boolean2()
    | list2(Type \type)
    | map2(Type key, Type v)
    | artifact(str name)
    | repository(str name)
    | selfie()
    ;

public data PhpOptionExpr = phpSomeExpr(PhpExpr expr) |
    ↪ phpNoExpr();

public data PhpOptionName = phpSomeName(PhpName name) |
    ↪ phpNoName();
```

```

public data PhpOptionElse = phpSomeElse(PhpElse e) |
     ↪ phpNoElse();

public data PhpActualParameter
    = phpActualParameter(PhpExpr expr, bool byRef)
    | phpActualParameter2(PhpExpr expr, bool byRef, bool
         ↪ isVariadic);

public data PhpConst = phpConst(str name, PhpExpr
     ↪ constValue);

public data PhpArrayElement = phpArrayElement(
     ↪ PhpOptionExpr key, PhpExpr val, bool byRef);

public data PhpName = phpName(str name);

public data PhpNameOrExpr = phpName2(PhpName name) |
     ↪ phpExpr(PhpExpr expr);

public data PhpCastType = phpIntCast() | phpBoolCast() |
     ↪ phpStringCast() | phpArrayCast() | phpObjectCast
     ↪ () | phpUnsetCast();

public data PhpClosureUse = phpClosureUse(PhpExpr varName
     ↪ , bool byRef);

public data PhpIncludeType = phpInclude() |
     ↪ phpIncludeOnce() | phpRequire() | phpRequireOnce
     ↪ ();

public data PhpExpr
    = phpArray(list[PhpArrayElement] items)
    | phpFetchArrayDim(PhpExpr var, PhpOptionExpr dim)
    | phpFetchClassConst(PhpNameOrExpr className, str
         ↪ constantName)
    | phpAssign(PhpExpr assignTo, PhpExpr assignExpr)
    | phpAssignWOp(PhpExpr assignTo, PhpExpr assignExpr,
         ↪ PhpOp operation)
    | phpListAssign(list[PhpOptionExpr] assignsTo,
         ↪ PhpExpr assignExpr)
    | phpRefAssign(PhpExpr assignTo, PhpExpr assignExpr)
    | phpBinaryOperation(PhpExpr left, PhpExpr right,
         ↪ PhpOp operation)
    | phpUnaryOperation(PhpExpr operand, PhpOp operation)
    | phpNew(PhpNameOrExpr className, list[
         ↪ PhpActualParameter] parameters)
    | phpCast(PhpCastType castType, PhpExpr expr)
    | phpClone(PhpExpr expr)
    | phpClosure(list[PhpStmt] statements, list[PhpParam]
         ↪  params, list[PhpClosureUse] closureUses,
         ↪ bool byRef, bool static)
    | phpFetchConst(PhpName name)
    | phpEmpty(PhpExpr expr)
    | phpSuppress(PhpExpr expr)
    | phpEval(PhpExpr expr)
```

```
   | phpExit(PhpOptionExpr exitExpr)
   | phpCall(PhpNameOrExpr funName, list[
     ↪ PhpActualParameter] parameters)
   | phpMethodCall(PhpExpr target, PhpNameOrExpr
     ↪ methodName, list[PhpActualParameter]
     ↪ parameters)
   | phpStaticCall(PhpNameOrExpr staticTarget,
     ↪ PhpNameOrExpr methodName, list[
     ↪ PhpActualParameter] parameters)
   | phpIncludeExpr(PhpExpr expr, PhpIncludeType
     ↪ includeType)
   | phpInstanceOf(PhpExpr expr, PhpNameOrExpr toCompare
     ↪ )
   | phpIsSet(list[PhpExpr] exprs)
   | phpPrint(PhpExpr expr)
   | phpPropertyFetch(PhpExpr target, PhpNameOrExpr
     ↪ propertyName)
   | phpShellExec(list[PhpExpr] parts)
   | phpTernary(PhpExpr cond, PhpOptionExpr ifBranch,
     ↪ PhpExpr elseBranch)
   | phpStaticPropertyFetch(PhpNameOrExpr className,
     ↪ PhpNameOrExpr propertyName)
   | phpScalar(PhpScalar scalarVal)
   | phpVar(PhpNameOrExpr varName)
   | phpYield(PhpOptionExpr keyExpr, PhpOptionExpr
     ↪ valueExpr)
   | phpListExpr(list[PhpOptionExpr] listExprs)
   | phpBracket(PhpOptionExpr bracketExpr)
   ;

public data PhpOp = phpBitwiseAnd() | phpBitwiseOr() |
   ↪ phpBitwiseXor() | phpConcat() | phpDiv()
        | phpMinus() | phpMod() | phpMul() |
          ↪ phpPlus() | phpRightShift() |
          ↪ phpLeftShift()
        | phpBooleanAnd() | phpBooleanOr() |
          ↪ phpBooleanNot() | phpBitwiseNot()
        | phpGt() | phpGeq() | phpLogicalAnd() |
          ↪ phpLogicalOr() | phpLogicalXor()
        | phpNotEqual() | phpNotIdentical() |
          ↪ phpPostDec() | phpPreDec() |
          ↪ phpPostInc()
        | phpPreInc() | phpLt() | phpLeq() |
          ↪ phpUnaryPlus() | phpUnaryMinus()
        | phpEqual() | phpIdentical() ;

public data PhpParam
   = phpParam(str paramName, PhpOptionExpr paramDefault,
     ↪  PhpOptionName paramType, bool byRef, bool
     ↪ isVariadic)
   ;

public data PhpScalar
   = phpClassConstant()
   | phpDirConstant()
   | phpFileConstant()
```

```
	| phpFuncConstant()
	| phpLineConstant()
	| phpMethodConstant()
	| phpNamespaceConstant()
	| phpTraitConstant()
	| phpNull()
	| phpInteger(int intVal)
	| phpString(str strVal)
	| phpBoolean(bool boolVal)
	| phpEncapsed(list[PhpExpr] parts)
	| phpEncapsedStringPart(str strVal)
	;

public data PhpStmt
	= phpBreak(PhpOptionExpr breakExpr)
	| phpClassDef(PhpClassDef classDef)
	| phpConsts(list[PhpConst] consts)
	| phpContinue(PhpOptionExpr continueExpr)
	| phpDeclare(list[PhpDeclaration] decls, list[PhpStmt
	    ↪ ] body)
	| phpDo(PhpExpr cond, list[PhpStmt] body)
	| phpEcho(list[PhpExpr] exprs)
	| phpExprstmt(PhpExpr expr)
	| phpFor(list[PhpExpr] inits, list[PhpExpr] conds,
	    ↪ list[PhpExpr] exprs, list[PhpStmt] body)
	| phpForeach(PhpExpr arrayExpr, PhpOptionExpr keyvar,
	    ↪  bool byRef, PhpExpr asVar, list[PhpStmt]
	    ↪ body)
	| phpFunction(str name, bool byRef, list[PhpParam]
	    ↪ params, list[PhpStmt] body, PhpOptionName
	    ↪ returnType)
	| phpGlobal(list[PhpExpr] exprs)
	| phpGoto(str label)
	| phpHaltCompiler(str remainingText)
	| phpIf(PhpExpr cond, list[PhpStmt] body, list[
	    ↪ PhpElseIf] elseIfs, PhpOptionElse elseClause)
	| phpInlineHTML(str htmlText)
	| phpInterfaceDef(PhpInterfaceDef interfaceDef)
	| phpTraitDef(PhpTraitDef traitDef)
	| phpLabel(str labelName)
	| phpNamespace(PhpOptionName nsName, list[PhpStmt]
	    ↪ body)
	| phpNamespaceHeader(PhpName namespaceName)
	| phpReturn(PhpOptionExpr returnExpr)
	| phpStaticVars(list[PhpStaticVar] vars)
	| phpSwitch(PhpExpr cond, list[PhpCase] cases)
	| phpThrow(PhpExpr expr)
	| phpTryCatch(list[PhpStmt] body, list[PhpCatch]
	    ↪ catches)
	| phpTryCatchFinally(list[PhpStmt] body, list[
	    ↪ PhpCatch] catches, list[PhpStmt] finallyBody)
	| phpUnset(list[PhpExpr] unsetVars)
	| phpUseExpr(set[PhpUse] uses)
	| phpWhile(PhpExpr cond, list[PhpStmt] body)
	| phpEmptyStmt()
	| phpBlock(list[PhpStmt] body)
```

```
    | phpNewLine()
    ;

public data PhpDeclaration = phpDeclaration(str key,
    ↪ PhpExpr val);

public data PhpCatch = phpCatch(PhpName xtype, str
    ↪ varName, list[PhpStmt] body);

public data PhpCase = phpCase(PhpOptionExpr cond, list[
    ↪ PhpStmt] body);

public data PhpElseIf = phpElseIf(PhpExpr cond, list[
    ↪ PhpStmt] body);

public data PhpElse = phpElse(list[PhpStmt] body);

public data PhpUse = phpUse(PhpName importName,
    ↪ PhpOptionName asName);

public data PhpClassItem
    = phpPropertyCI(set[PhpModifier] modifiers, list[
        ↪ PhpProperty] prop)
    | phpConstCI(list[PhpConst] consts)
    | phpMethod(str name, set[PhpModifier] modifiers,
        ↪ bool byRef, list[PhpParam] params, list[
        ↪ PhpStmt] body, PhpOptionName returnType)
    | phpTraitUse(list[PhpName] traits, list[
        ↪ PhpAdaptation] adaptations)
    ;

public data PhpAdaptation
    = phpTraitAlias(PhpOptionName traitName, str methName
        ↪ , set[PhpModifier] newModifiers,
        ↪ PhpOptionName newName)
    | phpTraitPrecedence(PhpOptionName traitName, str
        ↪ methName, set[PhpName] insteadOf)
    ;

public data PhpProperty = phpProperty(str propertyName,
    ↪ PhpOptionExpr defaultValue);

public data PhpModifier = phpPublic() | phpPrivate() |
    ↪ phpProtected() | phpStatic() | phpAbstract() |
    ↪ phpFinal();

public data PhpClassDef = phpClass(str className,
                                   set[PhpModifier] modifiers,
                                   PhpOptionName extends,
                                   list[PhpName] implements,
                                   list[PhpClassItem] members);

public data PhpInterfaceDef = phpInterface(str
    ↪ interfaceName,
                                           list[PhpName] extends
                                               ↪ ,
                                    list[PhpClassItem] members);

public data PhpTraitDef = phpTrait(str traitName, list[PhpClassItem] members);

public data PhpStaticVar = phpStaticVar(str name, PhpOptionExpr defaultValue);

public data PhpScript = phpScript(list[PhpStmt] body) | phpErrscript(str err);

public data PhpAnnotation
    = phpAnnotation(str key)
    | phpAnnotation(str key, PhpAnnotation v)
    | phpAnnotationVal(map[str k, PhpAnnotation v])
    | phpAnnotationVal(str string)
    | phpAnnotationVal(int integer)
    | phpAnnotationVal(bool boolean)
    | phpAnnotationVal(list[PhpAnnotation] items)
    | phpAnnotationVal(PhpAnnotation v)
;

// Changed because of lack of string abstractions of built-in library functions
public str toLowerCaseFirstChar(str text) = text;

PhpExpr toPhpExpr(Expression expr) {
    switch(expr) {

        // literals
        case integer(int i): return phpScalar(phpInteger(i));
        case string(str s): return phpScalar(phpString(s));
        case boolean(bool b): return phpScalar(phpBoolean(b));

        // arrays
        case \list(list[Expression] items): {
         list[PhpArrayElement] phpItems = [];
         for (i <- items)
            phpItems = phpItems + [phpArrayElement(phpNoExpr(), toPhpExpr(i), false)];
         return phpNew(phpName2(phpName("Vector")),
                    [phpActualParameter(phpArray(phpItems), false)]);
         }

        case arrayAccess(Expression variable, Expression arrayIndexKey):
            return phpFetchArrayDim(toPhpExpr(variable), phpSomeExpr(toPhpExpr(arrayIndexKey)));
        case \map(map[Expression key, Expression \value] m): {
            list[PhpActualParameter] elements = [];
            for (k <- m)
                elements = elements +
                            [phpActualParameter(phpNew(
                                ↪ phpName2(phpName("
                                ↪ Pair")),
                                [phpActualParameter(
                                    ↪ toPhpExpr(k),
                                    ↪ false),
                                    phpActualParameter(
                                        ↪ toPhpExpr(m[k
                                        ↪ ]), false)]),
                                        ↪  false)];
            return phpStaticCall(phpName2(phpName("
                ↪ MapFactory")), phpName2(phpName("
                ↪ createFromPairs")), elements);
        }

        case get(artifact(str name)):
            return phpPropertyFetch(phpVar(phpName2(
                ↪ phpName("this"))), phpName2(phpName("
                ↪ _" + toLowerCaseFirstChar(name))));

        case variable(str name):
            return phpVar(phpName2(phpName(name)));

        case ternary(Expression condition, Expression
            ↪ ifThen, Expression \else):
            return phpTernary(toPhpExpr(condition),
                ↪ phpSomeExpr(toPhpExpr(ifThen)),
                ↪ toPhpExpr(\else));

        case new(str artifact, list[Expression] args): {
            list[PhpActualParameter] phpParams = [];
            for (arg <- args)
                phpParams = phpParams + [
                    ↪ phpActualParameter(toPhpExpr(arg)
                    ↪ , false)];
            phpNew(phpName2(phpName(artifact)), phpParams
                ↪ );
        }

        // Binary operations
        case equals(Expression l, Expression r):
            return phpBinaryOperation(toPhpExpr(l),
                ↪ toPhpExpr(r), phpIdentical());
        case greaterThan(Expression l, Expression r):
            return phpBinaryOperation(toPhpExpr(l),
                ↪ toPhpExpr(r), phpGt());
        case product(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpMul());
        case remainder(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpMod());
        case division(Expression lhs, Expression rhs):
            return  phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpDiv());
        case addition(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpPlus());
        case subtraction(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpMinus());
        case \bracket(Expression e):
            return phpBracket(phpSomeExpr(toPhpExpr(e)));
        case greaterThanOrEq(Expression lhs, Expression
            ↪ rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpGeq());
        case lessThanOrEq(Expression lhs, Expression rhs)
            ↪ :
            return  phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpLeq());
        case lessThan(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpLt());
        case greaterThan(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpGt());
        case equals(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpIdentical());
        case nonEquals(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpNotIdentical());
        case and(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpLogicalAnd());
        case or(Expression lhs, Expression rhs):
            return phpBinaryOperation(toPhpExpr(lhs),
                ↪ toPhpExpr(rhs), phpLogicalOr());


        // Unary operations
        case negative(Expression e):
            return phpUnaryOperation(toPhpExpr(e),
                ↪ phpUnaryMinus());
        case positive(Expression e):
            return phpUnaryOperation(toPhpExpr(e),
                ↪ phpUnaryPlus());

        case invoke(str methodName, list[Expression] args
            ↪ ): {
            list[PhpActualParameter] phpParams = [];
            for (arg <- args)
                phpParams = phpParams + [
                    ↪ phpActualParameter(toPhpExpr(arg)
                    ↪ , false)];
            return phpMethodCall(phpVar(phpName2(phpName(
                ↪ "this"))), phpName2(phpName(
                ↪ methodName)), phpParams);
        }

        case invoke2(Expression prev, str methodName,
            ↪ list[Expression] args): {
            list[PhpActualParameter] phpParams = [];
            for (arg <- args)
                phpParams = phpParams +  [
                    ↪ phpActualParameter(toPhpExpr(arg)
                    ↪ , false)];

            return phpMethodCall(toPhpExpr(prev),
                ↪ phpName2(phpName(methodName)),
                ↪ phpParams);
        }


        // Property fetch
        case fieldAccess(str name):
            return phpPropertyFetch(phpVar(phpName2(
                ↪ phpName("this"))), phpName2(phpName(
                ↪ name)));

        case fieldAccess2(Expression prev, str name):
            return phpPropertyFetch(toPhpExpr(prev),
                ↪ phpName2(phpName(name)));

        case this():
            return phpVar(phpName2(phpName("this")));

        case _: throw unsupported();
    };
}
```

Appendix H

Rascal Verification Output

Negation Normal Form:

```
refine Formula#19 = atom(string)
refine Formula#2045 = and(Formula#2045, Formula#2045) |
    ↪ atom(string) | neg(Formula#19) | or(Formula#2045,
    ↪  Formula#2045)

result:
 success Formula#2045
store:
 [phi -> Formula]
```

Rename Struct Field

```
refine Expr#564 = fieldaccessexpr(Expr#564, Nominal#442)
    ↪ | functioncallexpr(Expr#564, string, list[Expr
    ↪ #564]) | varexpr(string)
refine Function#6215 = function(string, string, list[
    ↪ Parameter], Stmt#3305)
refine Nominal#442 = nfn() | other()
refine Nominal#nfn = nfn()
refine Nominal#ofn = ofn()
refine Package#23593 = package(map[string, Struct], map[
    ↪ string, Function#6215])
refine Stmt#3305 = assignstmt(Expr#564, Expr#564) | block
    ↪ (list[Stmt#3305]) | ifstmt(Expr#564, Stmt#3305,
    ↪ Stmt#3305) | returnstmt(Expr#564)

result:
 throw NoKey
store:
 [st -> string,pkg -> Package,newFieldName -> Nominal#nfn
    ↪ ,oldFieldName -> Nominal#ofn]

result:
 success Package#23593
store:
 [st -> string,pkg -> Package,newFieldName -> Nominal#nfn
    ↪ ,oldFieldName -> Nominal#ofn]
```

Desugar Oberon-0

```
refine ElseIf#104075 = elseif(Expression, list[Statement
    ↪ #104076])
refine Module#207238 = mod(Ident, Declarations, list[
    ↪ Statement#104076], Ident)
refine Statement#104076 = assign(Ident, Expression) |
    ↪ begin(list[Statement#104076]) | ifThen(Expression
    ↪ , list[Statement#104076], list[ElseIf#104075],
    ↪ list[Statement#104076]) | skip() | whileDo(
    ↪ Expression, list[Statement#104076])

result:
 error // Due to over-approximation
store:
 [mod -> Module]

result:
 success Module#207238
store:
 [mod -> Module#207238]
```

Glagol-to-PHP Simple Expressions:

```
refine Expression#0 = addition(Expression#0, Expression
    ↪ #0) | and(Expression#0, Expression#0) | boolean(
    ↪ Bool) | bracket(Expression#0) | division(
    ↪ Expression#0, Expression#0) | emptyExpr() |
    ↪ equals(Expression#0, Expression#0) | greaterThan(
    ↪ Expression#0, Expression#0) | greaterThanOrEq(
    ↪ Expression#0, Expression#0) | integer(int) |
    ↪ lessThan(Expression#0, Expression#0) |
    ↪ lessThanOrEq(Expression#0, Expression#0) |
    ↪ negative(Expression#0) | nonEquals(Expression#0,
    ↪ Expression#0) | or(Expression#0, Expression#0) |
    ↪ positive(Expression#0) | product(Expression#0,
    ↪ Expression#0) | remainder(Expression#0,
    ↪ Expression#0) | string(string) | subtraction(
    ↪ Expression#0, Expression#0) | ternary(Expression
    ↪ #0, Expression#0, Expression#0) | variable(string
    ↪ )
refine PhpExpr#187 = phpBracket(PhpOptionExpr#185) |
    ↪ phpScalar(PhpScalar#90) | phpTernary(PhpExpr#187,
    ↪  PhpOptionExpr#185, PhpExpr#187) |
    ↪ phpUnaryOperation(PhpExpr#187, PhpOp#160) |
    ↪ phpVar(PhpNameOrExpr#18)
refine PhpExpr#9207 = phpBracket(PhpOptionExpr#9206) |
    ↪ phpScalar(PhpScalar#90) | phpUnaryOperation(
    ↪ PhpExpr#9207, PhpOp#160) | phpVar(PhpNameOrExpr
    ↪ #18)
refine PhpExpr#9794 = phpBinaryOperation(PhpExpr#9794,
    ↪ PhpExpr#9794, PhpOp#9795) | phpBracket(
    ↪ PhpOptionExpr#9796) | phpScalar(PhpScalar#90) |
    ↪ phpUnaryOperation(PhpExpr#9794, PhpOp#160) |
    ↪ phpVar(PhpNameOrExpr#18)
refine PhpExpr#9810 = phpBinaryOperation(PhpExpr#9794,
    ↪ PhpExpr#9794, PhpOp#9795) | phpBracket(
    ↪ PhpOptionExpr#9206) | phpScalar(PhpScalar#90) |
    ↪ phpTernary(PhpExpr#187, PhpOptionExpr#185,
    ↪ PhpExpr#187) | phpUnaryOperation(PhpExpr#9207,
    ↪ PhpOp#160) | phpVar(PhpNameOrExpr#18)
refine PhpNameOrExpr#18 = phpName2(PhpName)
refine PhpOp#160 = phpUnaryMinus() | phpUnaryPlus()
refine PhpOp#9795 = phpDiv() | phpGeq() | phpGt() |
    ↪ phpIdentical() | phpLeq() | phpLogicalAnd() |
    ↪ phpLogicalOr() | phpLt() | phpMinus() | phpMod()
    ↪ | phpMul() | phpNotIdentical() | phpPlus()
refine PhpOptionExpr#185 = phpSomeExpr(PhpExpr#187)
refine PhpOptionExpr#9206 = phpSomeExpr(PhpExpr#9207)
refine PhpOptionExpr#9796 = phpSomeExpr(PhpExpr#9794)
refine PhpScalar#90 = phpBoolean(Bool) | phpInteger(int)
    ↪ | phpString(string)
result:
 throw Unsuppported
store:
 [expr -> Expression#0]

result:
 error // Due to over-approximation
store:
 [expr -> Expression#0]

result:
 success PhpExpr#9810
store:
 [expr -> Expression#0]
```

Glagol-to-PHP No Unary Expression

```
refine Bool#130 = false()
refine Expression#0 = addition(Expression#0, Expression
    ↪ #0) | and(Expression#0, Expression#0) |
    ↪ arrayAccess(Expression#0, Expression#0) | boolean
    ↪ (Bool) | bracket(Expression#0) | division(
    ↪ Expression#0, Expression#0) | emptyExpr() |
    ↪ equals(Expression#0, Expression#0) | fieldAccess(
    ↪ string) | fieldAccess2(Expression#0, string) |
    ↪ get(Type) | greaterThan(Expression#0, Expression
    ↪ #0) | greaterThanOrEq(Expression#0, Expression#0)
    ↪  | integer(int) | invoke(string, list[Expression
    ↪ #0]) | invoke2(Expression#0, string, list[
    ↪ Expression#0]) | lessThan(Expression#0,
    ↪ Expression#0) | lessThanOrEq(Expression#0,
    ↪ Expression#0) | list(list[Expression#0]) | map(
    ↪ map[Expression#0, Expression#0]) | new(string,
    ↪ list[Expression#0]) | nonEquals(Expression#0,
    ↪ Expression#0) | or(Expression#0, Expression#0) |
    ↪ product(Expression#0, Expression#0) | remainder(
    ↪ Expression#0, Expression#0) | string(string) |
    ↪ subtraction(Expression#0, Expression#0) | ternary
    ↪ (Expression#0, Expression#0, Expression#0) | this
    ↪ () | variable(string)
refine PhpActualParameter#2847 = phpActualParameter(
    ↪ PhpExpr#2848, Bool#130)
```

```
refine PhpActualParameter#2850 = phpActualParameter(
    ↪ PhpExpr#2852, Bool#130)
refine PhpActualParameter#37391 = phpActualParameter(
    ↪ PhpExpr#37393, Bool#130)
refine PhpActualParameter#37395 = phpActualParameter(
    ↪ PhpExpr#37388, Bool#130)
refine PhpActualParameter#38501 = phpActualParameter(
    ↪ PhpExpr#38510, Bool#130)
refine PhpActualParameter#38505 = phpActualParameter(
    ↪ PhpExpr#38509, Bool#130)
refine PhpActualParameter#54145 = phpActualParameter(
    ↪ PhpExpr#54146, Bool#130)
refine PhpActualParameter#54148 = phpActualParameter(
    ↪ PhpExpr#54142, Bool#130)
refine PhpActualParameter#54658 = phpActualParameter(
    ↪ PhpExpr#54659, Bool#130)
refine PhpActualParameter#54661 = phpActualParameter(
    ↪ PhpExpr#54655, Bool#130)
refine PhpActualParameter#598 = phpActualParameter(
    ↪ PhpExpr#607, Bool#130)
refine PhpActualParameter#602 = phpActualParameter(
    ↪ PhpExpr#603, Bool#130)
refine PhpActualParameter#62410 = phpActualParameter(
    ↪ PhpExpr#62411, Bool#130)
refine PhpActualParameter#62412 = phpActualParameter(
    ↪ PhpExpr#62417, Bool#130)
refine PhpActualParameter#62413 = phpActualParameter(
    ↪ PhpExpr#62414, Bool#130)
refine PhpArrayElement#2845 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#2848, Bool#130)
refine PhpArrayElement#37389 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#37393, Bool#130)
refine PhpArrayElement#38503 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#38509, Bool#130)
refine PhpArrayElement#54143 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#54146, Bool#130)
refine PhpArrayElement#54660 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#54655, Bool#130)
refine PhpArrayElement#600 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#603, Bool#130)
refine PhpArrayElement#62415 = phpArrayElement(
    ↪ PhpOptionExpr#483, PhpExpr#62417, Bool#130)
refine PhpExpr#2848 = phpBracket(PhpOptionExpr#2842) |
    ↪ phpMethodCall(PhpExpr#39, PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#2847]) | phpNew(PhpNameOrExpr
    ↪ #38, list[PhpActualParameter#2850]) |
    ↪ phpPropertyFetch(PhpExpr#39, PhpNameOrExpr#38) |
    ↪ phpScalar(PhpScalar#140) | phpStaticCall(
    ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[void]) |
    ↪  phpTernary(PhpExpr#2848, PhpOptionExpr#2842,
    ↪ PhpExpr#2848) | phpVar(PhpNameOrExpr#38)
refine PhpExpr#2852 = phpArray(list[PhpArrayElement
    ↪ #2845])
refine PhpExpr#37388 = phpArray(list[PhpArrayElement
    ↪ #37389])
refine PhpExpr#37393 = phpBracket(PhpOptionExpr#37386) |
    ↪ phpMethodCall(PhpExpr#39, PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#37391]) | phpNew(PhpNameOrExpr
    ↪ #38, list[PhpActualParameter#37395]) |
    ↪ phpPropertyFetch(PhpExpr#37393, PhpNameOrExpr#38)
    ↪  | phpScalar(PhpScalar#140) | phpStaticCall(
    ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[void]) |
    ↪  phpVar(PhpNameOrExpr#38)
refine PhpExpr#38509 = phpBracket(PhpOptionExpr#38500) |
    ↪ phpMethodCall(PhpExpr#38509, PhpNameOrExpr#38,
    ↪ list[PhpActualParameter#38505]) | phpNew(
    ↪ PhpNameOrExpr#38, list[PhpActualParameter#38501])
    ↪  | phpPropertyFetch(PhpExpr#39, PhpNameOrExpr#38)
    ↪  | phpScalar(PhpScalar#140) | phpStaticCall(
    ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[void]) |
    ↪  phpVar(PhpNameOrExpr#38)
refine PhpExpr#38510 = phpArray(list[PhpArrayElement
    ↪ #38503])
refine PhpExpr#39 = phpVar(PhpNameOrExpr#38)
refine PhpExpr#54142 = phpArray(list[PhpArrayElement
    ↪ #54143])
refine PhpExpr#54146 = phpBracket(PhpOptionExpr#54140) |
    ↪ phpMethodCall(PhpExpr#39, PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#54145]) | phpNew(PhpNameOrExpr
    ↪ #38, list[PhpActualParameter#54148]) |
    ↪ phpPropertyFetch(PhpExpr#39, PhpNameOrExpr#38) |
    ↪ phpScalar(PhpScalar#140) | phpStaticCall(
    ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[void]) |
    ↪  phpVar(PhpNameOrExpr#38)
refine PhpExpr#54655 = phpBinaryOperation(PhpExpr#54655,
    ↪ PhpExpr#54655, PhpOp#54656) | phpBracket(
    ↪ PhpOptionExpr#54657) | phpMethodCall(PhpExpr#39,
    ↪ PhpNameOrExpr#38, list[PhpActualParameter#54661])
    ↪  | phpNew(PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#54658]) | phpPropertyFetch(
    ↪ PhpExpr#39, PhpNameOrExpr#38) | phpScalar(
    ↪ PhpScalar#140) | phpStaticCall(PhpNameOrExpr#38,
    ↪ PhpNameOrExpr#38, list[void]) | phpVar(
    ↪ PhpNameOrExpr#38)
refine PhpExpr#54659 = phpArray(list[PhpArrayElement
    ↪ #54660])
refine PhpExpr#603 = phpBracket(PhpOptionExpr#597) |
    ↪ phpFetchArrayDim(PhpExpr#603, PhpOptionExpr#597)
    ↪ | phpMethodCall(PhpExpr#39, PhpNameOrExpr#38,
    ↪ list[PhpActualParameter#602]) | phpNew(
    ↪ PhpNameOrExpr#38, list[PhpActualParameter#598]) |
    ↪  phpPropertyFetch(PhpExpr#39, PhpNameOrExpr#38) |
    ↪  phpScalar(PhpScalar#140) | phpStaticCall(
    ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[void]) |
    ↪  phpVar(PhpNameOrExpr#38)
refine PhpExpr#607 = phpArray(list[PhpArrayElement#600])
refine PhpExpr#62411 = phpNew(PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#62412])
refine PhpExpr#62414 = phpArray(list[PhpArrayElement
    ↪ #62415])
refine PhpExpr#62417 = phpBracket(PhpOptionExpr#62409) |
    ↪ phpMethodCall(PhpExpr#39, PhpNameOrExpr#38, list[
    ↪ PhpActualParameter#62412]) | phpNew(PhpNameOrExpr
```

```
      ↪ #38, list[PhpActualParameter#62413]) |
      ↪ phpPropertyFetch(PhpExpr#39, PhpNameOrExpr#38) |
      ↪ phpScalar(PhpScalar#140) | phpStaticCall(
      ↪ PhpNameOrExpr#38, PhpNameOrExpr#38, list[
      ↪ PhpActualParameter#62410]) | phpVar(PhpNameOrExpr
      ↪ #38)
refine PhpExpr#90001 = phpBinaryOperation(PhpExpr#54655,
      ↪ PhpExpr#54655, PhpOp#54656) | phpBracket(
      ↪ PhpOptionExpr#54140) | phpFetchArrayDim(PhpExpr
      ↪ #603, PhpOptionExpr#597) | phpMethodCall(PhpExpr
      ↪ #38509, PhpNameOrExpr#38, list[PhpActualParameter
      ↪ #38505]) | phpNew(PhpNameOrExpr#38, list[
      ↪ PhpActualParameter#54148]) | phpPropertyFetch(
      ↪ PhpExpr#37393, PhpNameOrExpr#38) | phpScalar(
      ↪ PhpScalar#140) | phpStaticCall(PhpNameOrExpr#38,
      ↪ PhpNameOrExpr#38, list[PhpActualParameter#62410])
      ↪  | phpTernary(PhpExpr#2848, PhpOptionExpr#2842,
      ↪ PhpExpr#2848) | phpVar(PhpNameOrExpr#38)
refine PhpNameOrExpr#38 = phpName2(PhpName)
refine PhpOp#54656 = phpDiv() | phpGeq() | phpGt() |
      ↪ phpIdentical() | phpLeq() | phpLogicalAnd() |
      ↪ phpLogicalOr() | phpLt() | phpMinus() | phpMod()
      ↪ | phpMul() | phpNotIdentical() | phpPlus()
refine PhpOptionExpr#2842 = phpSomeExpr(PhpExpr#2848)
refine PhpOptionExpr#37386 = phpSomeExpr(PhpExpr#37393)
refine PhpOptionExpr#38500 = phpSomeExpr(PhpExpr#38509)
refine PhpOptionExpr#483 = phpNoExpr()
refine PhpOptionExpr#54140 = phpSomeExpr(PhpExpr#54146)
refine PhpOptionExpr#54657 = phpSomeExpr(PhpExpr#54655)
refine PhpOptionExpr#597 = phpSomeExpr(PhpExpr#603)
refine PhpOptionExpr#62409 = phpSomeExpr(PhpExpr#62417)
refine PhpScalar#140 = phpBoolean(Bool) | phpInteger(int)
      ↪  | phpString(string)

result:
 throw Unsuppported
store:
 [expr -> Expression#0]

result:
 error // Due to over approximation
store:
 [expr -> Expression#0]

result:
 success PhpExpr#90001?
store:
 [expr -> Expression#0]
```